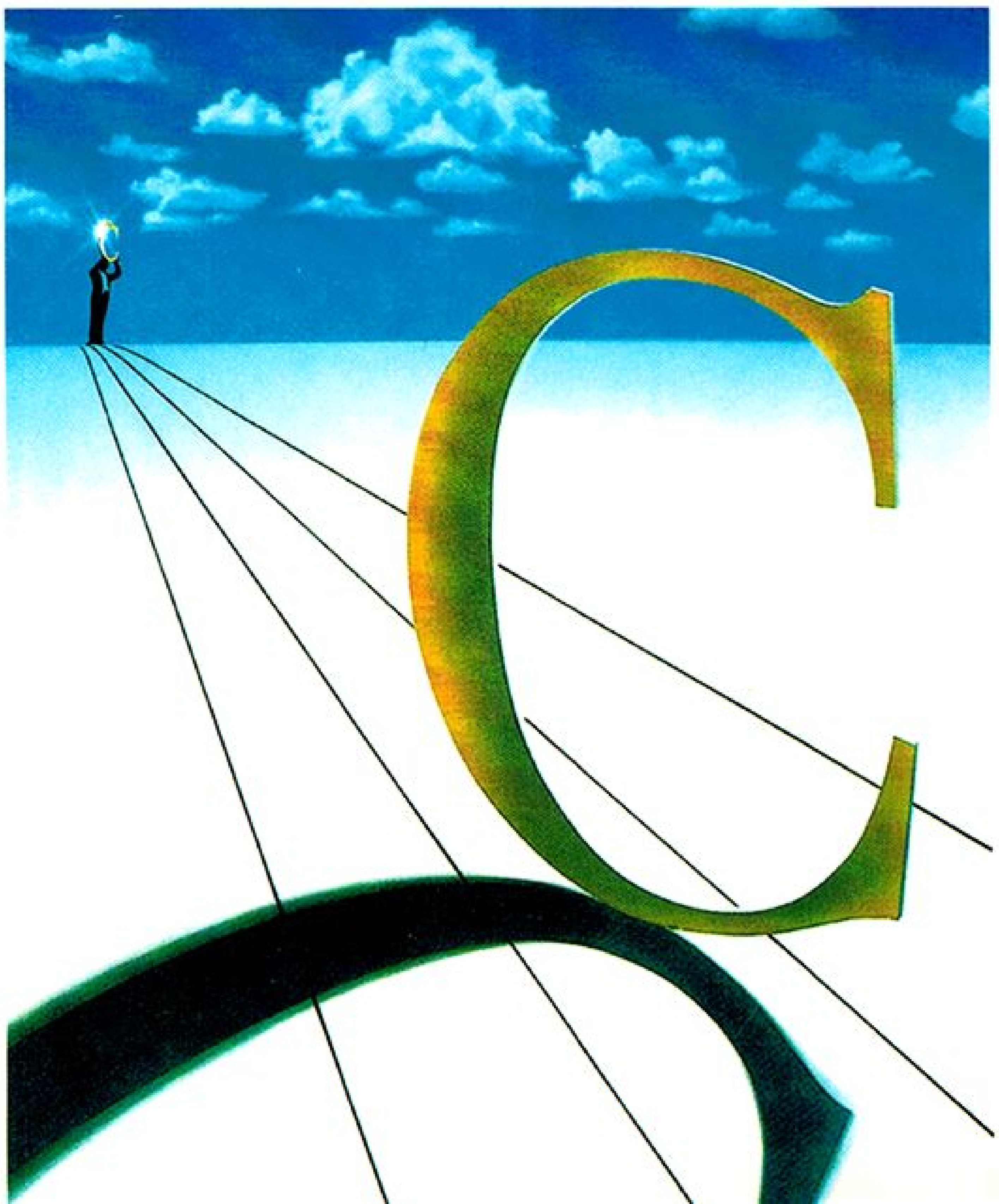
TURBO C®

USER'S GUIDE



BORLAND

TURBO C®

Borland's No-Nonsense License Statement!

This software is protected by both United States copyright law and international treaty provisions. Therefore, you must treat this software *just like a book*, with the following single exception. Borland International authorizes you to make archival copies of the software for the sole purpose of backing-up our software and protecting your investment from loss.

By saying, "just like a book," Borland means, for example, that this software may be used by any number of people and may be freely moved from one computer location to another, so long as there is **no possibility** of it being used at one location while it's being used at another. Just like a book that can't be read by two different people in two different places at the same time, neither can the software be used by two different people in two different places at the same time (unless, of course, Borland's copyright has been violated).

Programs that you write and compile using the Turbo C language compiler may be used, given away or sold without additional license or fees, as long as all copies of such programs bear a copyright notice. By "copyright notice" we mean either your own copyright notice or, if you prefer, the statement, "Created using Turbo C, Copyright © Borland 1987, 1988." Included in the Turbo C diskettes are several support files that contain encoded hardware and font information used by the standard graphics library (GRAPHICS.LIB). These files, which can be listed by typing DIR *.CHR and DIR *.BGI, are proprietary to Borland International. You may use these files with the programs you create with Turbo C for your own personal use. In addition, to the extent the programs you write and compile using the Turbo C language compiler make use of these support files, you may distribute these support files in combination with such programs, provided that you do not use, give away, or sell the support files separately, and all copies of such programs bear a copyright notice.

The sample programs included on the Turbo C diskettes provide a demonstration of how to use the various features of Turbo C. They are intended for educational purposes only. Borland International grants you (the registered owner of Turbo C) the right to edit or modify these sample programs for your own use, but you may not give away or sell them, alone or as part of any program, in executable, object or source code form. You may, however, incorporate miscellaneous sample program routines into your programs, as long as your resulting programs do not substantially duplicate all or part of a sample program in appearance or functionality and all copies of such programs bear a copyright notice.

Limited Warranty

With respect to the physical diskette and physical documentation enclosed herein, Borland International, Inc. ("Borland") warrants the same to be free of defects in materials and workmanship for a period of 60 days from the date of purchase. In the event of notification within the warranty period of defects in material or workmanship, Borland will replace the defective diskette or documentation. **If you need to return a product, call the Borland Customer Service Department to obtain a return authorization number.** The remedy for breach of this warranty shall be limited to replacement and shall not encompass any other damages, including but not limited to loss of profit, and special, incidental, consequential, or other similar claims.

Borland International, Inc. specifically **disclaims** all other warranties, expressed or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the diskette and documentation, and the program license granted herein in particular, and without limiting operation of the program license with respect to any particular application, use, or purpose. **In no event shall Borland be liable for any loss of profit or any other commercial damage, including but not limited to special, incidental, consequential or other damages.**

Governing Law

This statement shall be construed, interpreted, and governed by the laws of the state of California. Use, duplication, or disclosure by the U.S. Government of the computer software and documentation in this package shall be subject to the restricted rights under DFARS 52.227-7013 applicable to commercial computer software.

平成元年１０月２４日

<u>ユーザー各位</u>

（株）ボーランドジャパン

拝啓　皆様におかれましては、益々ご清栄のこととお慶び申し上げます。
平素は、弊社製品に格別のご愛顧を承り、まことにありがとうございます。
このたびお申し込みいただきました、Turbo シリーズのメンテナンスバージョンをお送りいたしますので、よろしくご査収ください。なお、修正内容につきましては、それぞれ下記のドキュメントファイルを参照してくださいますようお願い致します。
これらのメンテナンスバージョンを作成するにあたりまして、多くのユーザーの方々から、貴重なご助言、ご協力をいただきました。この場を借りまして、深くお礼申し上げます。

敬具

記

・Turbo C 2.0
修正内容については、マスターディスクの 1 枚目の TC20C.DOC に記載されております。INSTALL.EXE は、既存の Turbo C 2.0 のオプションを新しいバージョンに転送するモードが用意されています。（ハードディスクのみ対応）　ただし、ディレクトリ情報は転送されませんのでご注意ください。

・Turbo PASCAL 5.0
修正内容については、TP50B.DOC に記載されております。INSTALL.EXE は、既存のTurbo PASCAL 5.0 のオプションを新しいバージョンに転送するモードが用意されています。（ハードディスクのみ対応）　ただし、ディレクトリ情報は転送されませんのでご注意ください。

・Turbo Assembler & Debugger 1.0
修正内容については、TD10B.DOC に記載されております。Turbo Assembler については、TASM.EXE （およびドキュメントファイル）のみが変更されており、1枚のディスクに格納されています。（2HD の場合）

追記
ご案内いたしました修正内容のうち、Turbo C 2.0 について、#error の漢字対応は、別の不具合が発生する関係で修正が見送られました。修正内容が、ご案内したものと異なっている点につきまして、深くお詫び申し上げます。詳細につきましてはTC20C.DOC をご覧ください。

以上

Turbo C ®

ユーザーズガイド

バージョン 2.0

(NEC PC-9801 シリーズ)
(IBM PC, XT, AT, PS/2, および完全な互換機)

Borland International
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95066-0001

編訳 (株)マイクロソフトウェア アソシエイツ 1988

目　次

はじめに

Turbo Cは，高速で効率的なコンパイラを望んでいるCプログラマ，'Turbo'のよい環境の中でCを学びたいと考えているTurbo Pascalプログラマ，速くて使いやすい処理系でCを学びたいと考えている方たち，etc，すべてにうってつけのシステムです。

C言語は，構造的でモジュール化ができ，コンパイル可能な汎用のプログラミング言語であり，従来はシステムプログラミングに使用されてきたものです。可搬性にすぐれているので，Cで書かれたプログラムは容易に他の処理系に移行することができます。Cを使えば，ほとんどどんな種類の仕事でも行なうことができます。ただ，これまで存在していたCコンパイラの処理速度はとても遅く，使いにくいものが多かったのですが，Turbo Cの処理はとても速く，プログラムの吟味により多くの時間をさくことができるのです。

Turbo Cのパッケージ

購入されたTurbo Cのパッケージには，マスターディスクのセットと2冊のマニュアルTurbo Cユーザーズガイド（本書）とTurbo Cリファレンスガイドが含まれているはずです。マスターディスクの中には，Turbo Cを使ってプログラムを作り，コンパイル，リンク，実行する場合に必要となるすべてのプログラム，ファイルおよびライブラリが含まれています。パッケージにはさらに，いくつかのスタンドアロンのユーティリティ，文脈に即したヘルプファイル，統合デバッガ，および2冊のガイドではカバーしきれなかった最新の情報を含んだファイルも入っています。

このユーザーズガイドは，Cを初めて学ぶ人のためのハンドブックおよびガイドとして作成されています(もちろんベテランの復習にも適していますが)。また，Turbo Cエディタ，コマンドラインオプション，ユーティリティ（CPP,MAKE,TLINK,TILB,GREP,BGIOBJ,OBJXREF),Turbo Cのカスタマイズなどに関する情報も含んでいます。リファレンスガイドでは，Turbo Cの数多くのライブラリルーチンすべてについて詳細な説明がなされています。また，コンパイラのエラーメッセージ，Turbo Cの言語構文についてもリファレンスガイドにまとめられています。まだあまりCに慣れていないのであれば，リファレンスガイドを読み始める前にユーザーズガイドをまずよく読んでください。

Turbo C 2.0 の新しい機能

Turbo C 2.0では，新たに以下の機能が追加あるいは拡張されました。

- ■統合デバッグができるようになりました。これを使ってコードをステップ実行，トレースができます。ブレークポイントを設定できます。式をウオッチし，評価できます。
- ■バージョン1.5に比べ，コンパイル速度が20～30%向上しました。また，リンカもより高速になりました。
- ■エディタのバッファのためにEMS記憶が利用できます。これにより，コンパイルやプログラムの実行に使えるメモリが64Kバイト増えました。
- ■メモリ割り当て関数および文字列関数がより高速になりました。
- ■浮動小数点演算エミュレーションがより高速になりました。
- ■ **signal** 関数と **raise** 関数が新たに加わりました。
- ■コンパイル時に，プログラム中にマシン語のコードを挿入できる__**emit**__機能が加わりました。
- ■インストール可能なドライバやフォントなどを含めて，多くの関数が加わり，BGIグラフィックスライブラリがより高機能になりました。
- ■コマンドラインでワイルドカード展開がサポートされました。
- ■タイニィモデルのプログラムについて，リンカが.COMファイルを作成することができます。
- ■ボーランド社の新しいスタンドアロンのデバッガをサポートしています。
- ■MAKEユーティリティで依存関係の自動チェックを行ないます。
- ■long double 型の定数や変数をサポートしています。
- ■ブロックのインデント/逆インデントや最適フィルなど，新たな編集機能が加わりました。

動作環境

Turbo Cは，NEC PC-9801シリーズのコンピュータ（XLおよびVXのハイレゾリューションモード，XA，LTは除きます）の上で動作します。Turbo Cを走らせるには，バージョン2.0以上のMS-DOSと最低512KのRAMが必要です。高解像度のモニタディスプレイと1台のフロッピィディスクドライブがあればTurbo Cを動かすことができます。ただし，できればフロッピィディスクドライブ2台，あるいはフロッピィディスクドライブ1台とハードディスク，またRAMは640Kあったほうがよいでしょう。

IBM PCバージョンのTurbo Cは，XT，AT，PS/2などのIBM PCファミリィのコンピュータと，本当の意味での互換性があるIBM互換機の上で走ります。Turbo Cを走らせるには，バージョン2.0以上のDOSと，448K以上のRAMが必要です。モニタは，横80カラムのものであれば，どれでも対応できます。フロッピィドライブは1台あればTurbo Cを使えますが，2台のフロッピィドライブ，あるいはハードディスクと1台のフロッピィドライブの使用をおすすめします。

Turbo Cには，8087あるいは80287数値演算コプロセッサチップを利用するための浮動小数点ルーチンも含まれています。数値演算コプロセッサチップが搭載されていない場合は，Turbo Cはそのエミュレーションを行ないます。80x87チップを使うとプログラムの効率は大きく向上しますが，必須のものではありません。

Turbo C 処理系

Turbo Cは，米国規格協会（ANSI：American National Standard Institute）のC標準化案，およびB.W.カーニハンとD.M.リッチィによる定義を完全にサポートしています。さらに，複数の言語や異なるメモリモデルを混在させてプログラミングすることができるなどのオプションの拡張機能を備えており，PCの能力を遺憾なく発揮させることができます。

分冊 I：ユーザーズガイド

ユーザーズガイド（この本）では，まずTurbo Cとはどんなものであるかを紹介したあと，プログラムをどのように作成し，どのように走らせるかを示します。また，コンパイル，リンク，エラー原因の探し方，プロジェクト-Makeの使い方などの話題を理解する場合に必要となる基礎知識についても述べています。次に，このユーザーズガイドの各章で述べられている内容の要約を示します。

第1章「使い始める前に」では，TurboCを，使用するハードウェアにインストールする手順を説明しています。また，ユーザーズガイドの残りの部分をどのように読み進めていけばよいかも示しています。

第2章「さあ始めよう」では，Turbo Cの単純なプログラムをロードし，コンパイル，実行，編集，およびセーブするというTurbo C統合開発環境を使って行く上での基本事項を説明します。

第3章「**全部まとめて――コンパイルから実行まで**」では，Turbo Cの**Run**コマンドの使い方を示し，プログラムを構成するファイルをどのように"Make"（再作成）するかを説明しています。

第4章「**プログラムのデバッグ**」では，Turbo Cの統合デバッガを紹介し，例題プログラムに埋め込まれたバグの解析を通じて，デバッガのさまざまな機能を説明していきます。

第5章「**Turbo C 統合開発環境**」では，Turbo Cのテキストエディタ，統合デバッガ，メニューシステムについて説明し，またピックファイルとコンフィギュレーションファイルについても述べています。

第6章「**Turbo C でのプログラミング**」では，Turbo Cのプログラムを作って実行する際に必要となる基本的なステップについて説明し，しだいに短いながらやや高度なプログラム例へと進んでいきます。

第7章「**続：Turbo C でのプログラミング**」では，配列，ポインタ，構造体，制御構文，ストリームなどの少し進んだ要素を含んだプログラミングについて述べられます。

第 8 章「Turbo C ビデオ関数」では，最初にビデオモードとウィンドウについて述べ，次にテキストモードとグラフィックスモードでのプログラミングについて説明します。

第 9 章「Turbo Pascal プログラマのためのノート」では，Turbo C と Turbo Pascal のプログラム例を示して，両言語の違いについて説明し，また Turbo Pascal プログラマが犯しやすいまちがいについても取り上げています。

第 10 章「Turbo Prolog とのインターフェース」では，Turbo C で書かれたモジュールを，Turbo Prolog のプログラムと結びつけるにはどうしたらよいかを説明し，例を使ってその過程が示しています。

第 11 章「Turbo C 言語リファレンス」では，Turbo C が K&R の言語の定義と異なる点をすべて示しています。また，現在の ANSI 標準化案には含まれていない Turbo C の拡張について説明しています。

第 12 章「Turbo C での上級プログラミング」では，スタートアップコード，さまざまなメモリモデルでのメモリ構成，ポインタ演算，アセンブリ言語インターフェース，浮動小数点ライブラリの使い方などに関して詳しく説明しています。

付録 A「Turbo C エディタ」では，エディタのすべてのコマンドに関して説明されています。エディタのコマンドに関しては，第5章でも簡単に述べられています。

付録 B「TCC コマンドラインオプション」では，Turbo C でユーザが選択できるコンパイラオプションについて述べています。

付録 C「Turbo C ユーティリティ」では，Turbo C パッケージに含まれている以下の7つのユーティリティについて解説しています。CPP：プリプロセッサ，MAKE：プログラム作成ツール，TLINK：Turbo　Link ユーティリティ，TLIB：Turbo ライブラリアン，GREP：ファイル検索ユーティリティ，BGIOBJ：グラフィックスドライバ/フォント変換ユーティリティ，OBJXREF：オブジェクトモジュール・クロスリファレンサ。

付録 D「TCINST：Turbo C のカスタマイズ」では，Turbo C の統合開発環境 TC.EXE を，ユーザ専用のシステムにするためのプログラム TCINST の使い方を説明しています。

付録 E「MicroCalc」では，Turbo C のサンプルスプレッドシート MicroCalc のコンパイル手順と操作の概要を説明しています。

分冊 II：リファレンスガイド

リファレンスガイドは，経験のある C プログラマのために書かれたもので，言語および実行時環境に関する処理系特有の詳細について述べています。そのあと，Turbo C で使用できるすべての関数の定義が ABC 順で与えられています。リファレンスガイドは次のような構成になっています。

第 1 章「Turbo C ライブラリルーチンを使うには」では，Turbo C のインクルードファイル(.h)とカテゴリ別のライブラリルーチンの一覧，**main** 関数とその引数，および Turbo C のグローバル変数について解説しています。

第 2 章「Turbo C ライブラリ」では，Turbo C のすべてのライブラリ関数を ABC 順で示し，説明を加えます。各関数に対して，その形式，インクルードファイル，関連関数，機能の説明，戻り値，可搬性が示されています（なお，テキストライブラリ，グラフィックスルーチン，BIOS サポート関数，日本語処理関数などについては，節をわけてまとめてあります）。

付録 A「コンパイラエラーメッセージ」では，すべてのエラーメッセージについて説明し，考えられる原因も示しています。

付録 B「言語の構文の要約」では，修正 BNF（バッカスナウア記法）で Turbo C のすべての構文が述べられています。

推薦書籍

次に示す図書は，Turbo C のマニュアルの補足として役立つでしょう。

- ■最も広く認められている C に関する記述は"The C Programming Language", Kernighan & Ritchie (New Jersey : Prentice-Hall,1978) です（邦訳「プログラミング言語 C」石田晴久訳，共立出版）。
- ■ ANSI の C 標準化委員会 X3J11は現在 C 言語の標準を作成中ですが，Turbo C は ANSI C の標準化案に準拠しています。

はじめて C 言語を学ぶのであれば，Turbo C を使って，カーニハン&リッチィの本の練習問題をこなしてゆくことをすすめます。すでに C の経験を充分つんだ人であれば，たやすく Turbo C を使いこなすことができるでしょう。

C と Turbo C に関する他の書籍については，このマニュアルの最後にある参考文献を参照してください。

マニュアル表記に関する規約（書体）

マニュアルの表記に際して使用される特殊なものとして次のようなものがあります。

[ ]　　書式や MS-DOS コマンドラインにおける大カッコ[ ]は，システムによって異なるオプションの入力やデータを囲むものです。そのとおりにカッコをタイプしてはいけません。

〈 〉　　関数の説明における不等号は，インクルードファイルを囲むのに使われます。

Boldface　　Turbo C の関数名（**printf** など）はこのようなボールド体で表記しています。

Gothic　　Turbo C の予約語（char,switch,near,cdecl など）はゴシック体で表わします。

Italic　　文章中の変数名やユーザ定義の関数名（*sum,get _ratio* など）はイタリック体で表記します。

Bold-italic　　Turbo C で定義されているグローバル変数（***_fmode,errno*** など）はこの書体で表記します。

KEYCAPS　　キーボード上のキー（***F10,GRPH-X,DEL*** など）はこの書体で表記します。

謝辞

このマニュアルでは，以下に示すいくつかの製品を参照しています。

製品に関するお問い合わせ

ソフトウェアおよびマニュアルに関して，ご質問・ご意見などがありましたら，下記の**MSA カスタマーサポートセンター**まで，お電話，書面またはファクシミリにてお問い合わせください。なお，お電話での受付時間は平日の9：00am～12：00amおよび1：00pm～5：00pmとさせていただきます（土曜，日曜，祝日は休業とさせていただきますので御了承ください）。

〒107　東京都港区南青山7-8-1 小田急南青山ビル
（株）マイクロソフトウェアアソシエイツ
カスタマーサポートセンター
（TEL）03-486-1403
（FAX）03-486-8905

お問い合わせの際には，下記の事項をお伝えください。

- ■製品名とバージョン番号（Turbo C 2.0など）
- ■製品のシリアル番号（マスターディスクに記載されているもの）
- ■コンピュータ名とモデル番号（PC-9801VX など）
- ■オペレーティングシステムとバージョン番号（MS-DOS 3.1など）

なお，書面にてお問い合わせの際には，CONFIG.SYS および AUTOEXEC.BAT の内容，簡潔なプログラムリストを添付されるようお願い致します。

第1章

使い始める前に

Turbo C のパッケージには，2つのコンパイラが含まれています。1つは統合環境版，もう1つはスタンドアロンのコマンドライン版です。Turbo C を使用するシステムにインストールするには，マスターディスクから作業用フロッピィあるいはハードディスクにファイルをコピーします。この仕事は，マスターディスクにある INSTALL プログラムを使って，対話形式で行なうことができます。マスターディスクは，NEC PC-9801シリーズ（または IBM PC およびそのフル互換機）で読み込むことができる，MS-DOS のフォーマットで作成されています。マスターディスクに含まれているファイルの一覧は，マスターディスクの1枚目（Installation Disk）の中の README ファイルに書かれています。

注意：このマニュアルでは，ユーザがMS-DOS のコマンドの基本操作を理解していることを想定して書かれています。たとえば，FORMAT コマンドや DISKCOPY コマンドは，バックアップコピーを作成したり，作業ディスクを準備する際に使わなければなりません。MS-DOS のコマンドの使い方がまだよくわからない方は，Turbo C のセットアップを始める前に MS-DOS ユーザーズリファレンスマニュアルを参照してください。

パッケージを手にしたら，まず最初にマスターディスクのコピーを作成し，オリジナルのマスターディスクは安全な場所に保管しておいてください。マスターディスクから直接 Turbo C を起動しないようにしてください。なんらかの要因で作業用ファイルが壊れてしまった場合には，オリジナルのディスクを使わなくてはならないからです。

この章では……

この章では，README ファイルの読み方と Turbo C をコンピュータシステムにインストールする方法を述べています。また，使用する人の経験に基づいて次にマニュアルのどこへ進めばよいかを示しています。

README ファイル

まず最初にしなければならない一番重要なことは，Turbo C で何かを行なう前に，マスターディスク1枚目の中にある README ファイルを見ることです。このファイルには，マニュアルではカバーしきれなかった最新の情報が含まれています。さらに，この中にはマスターディスクに収められているすべてのファイルの一覧表が入っており，各ファイルの内容が簡単に述べてあります。

README ファイルを見るには，マスターディスクの1枚目（Installation Disk）をフロッピィドライブに入れ，そのドライブにログインして README とタイプし，**リターン**（***Enter***）を押してください。README に入ったら，上下の矢印キー（***Up,Down***）を使ってファイルの中を行ったり来たりすることができます。***ROLLUP/ROLLDOWN***（***PgUp/PgDn***）を使うと1画面単位でスクロールします。終了したいときには ***ESC*** を押してください。

Turbo C をシステムにインストールする

Turbo C パッケージには，統合環境版とコマンドライン版の両バージョンのコンパイラを使う場合に必要なすべてのファイル，プログラム，スタートアップコード，6つのモデルと8087コプロセッサのエミュレーションに対応するライブラリが含まれています。Turbo C を初めてインストールするときや，前のバージョン（1.5）からアップグレードしてインストールするときには，INSTALL プログラムを使うと簡単に行なえます。

フロッピィディスクシステムへのインストール

ハードディスクドライブがなく，1台あるいは2台のフロッピーディスクドライブがあるシステムへインストールする場合は，INSTALL を実行する前にフォーマット済みの空のディスクを何枚か用意してください。

INSTALL を1回実行するたびに，メモリモデル1つに対応する Turbo C をインストールすることができます。2つ以上のメモリモデルに対してインストールを実行したい場合は，インストールしたいメモリモデルごとに3枚のディスクを用意してください。

INSTALL を実行する

Turbo C のインストールプログラム INSTALL は，ユーザを導いてインストールを実行していきます。ユーザは，ただステップごとに画面に現われる指示に従っていけばよいのです。ただし，表示される指示は注意深く読んでください。

INSTALL は，マスターディスクをいつドライブに入れればよいかを指示してくれます。また，メニューやプロンプトを表示して入力を要求してくることもあります。

注意：プロンプトのいくつかにおいてはデフォルトが決まっており，メニューの下に[]で囲まれて表示されます。デフォルトを選択したい場合は，単に**リターン**（***Enter***）を押してください。デフォルトは，そのプロンプトに対するおすすめの選択といえます。

INSTALL は次の手順で実行します。

1. マスターディスクの1 of #（IBM PC バージョンでは Installation Disk）というラベルが貼ってあるマスターディスクをドライブ A に入れてください。
2. A：とタイプして，**リターン**（***Enter***）を押してください。
3. INSTALL とタイプして，**リターン**を押してください。

以降は，INSTALL プログラムが画面に表示してくる指示に従ってください。

INSTALL が終了すれば，すぐにでも Turbo C を使用できる状態になっているはずです。

注意：Turbo C の統合開発環境を試した後，いくつかのオプションをカスタマイズしたくなる場合がよくあります。TCINST というプログラムを使えば，それが容易にできます。それに関する説明は，このマニュアルの付録 D にあります。

ラップトップシステムでの Turbo C のセットアップ（IBM PC）

ラップトップスタイルのコンピュータ（LCD やプラズマディスプレイを備えたもの）の場合には，前の節で述べたインストールのステップのあとで，Turbo C を使う前にスクリーンパラメータを設定しておくとよいでしょう。Turbo C の統合開発環境（TC.EXE）は，起動する前に，DOS のコマンドラインから MODE BW80を入力しておくと，システムに最適な動作を示します。

MicroCalc の使い方

マスターディスクには，MicroCalc というスプレッドシートプログラムのソースコードが入っています。これをコンパイルして実行したい方は付録 E を見てください。

次にどこへ進めばよいか？

さて，Turbo C のインストールが済んだら，もうこのガイドを読み進めながら，実際にTurbo C を使い始める準備はできています。なお，このユーザーズガイドは，3種類の読者層を想定して作られており，各層のユーザの目的に沿うような章がいくつか用意されています。そこで，急いで先に進む前に，次の案内を眺めて，次にどの章を読めばよいか見つけてください。

C を学び始めたプログラマ

今まさに C 言語を学び始めたばかりの人は，第2章，第3章から始めるのがよいでしょう。この2つの章では，Turbo C 統合開発環境を紹介し，自分のプログラムを編集しセーブする方法に加えて，簡単な Turbo C プログラムのロード，コンパイル，リンク，および実行の手順を説明しています。第4章では，Turbo C 統合デバッガを紹介しています。次に第6章と第7章を読んでください。この2つの章では，C 言語のプログラミングにおける基本的な要素について入門者向けに解説しています。統合開発環境にまだあまり慣れないうちは，第5章を適宜参照する必要があるでしょう。第8章では，Turbo C の素晴らしいグラフィックス機能を紹介しています。

経験のある C プログラマ

C のプログラミングの経験のある方は，自分の書いたプログラムをこの処理系で動かすことにはそれほど困難はないでしょう。ただし，第11章「**Turbo C 言語リファレンス**」に目を通して，Turbo C がカーニハン&リッチィの定義や，ANSI 標準化案とどんな点が異なっているかを確認しておくとよいでしょう。Turbo C へのプログラムの移植や，Turbo C でプログラム開発に，すぐにでもとりかかろうとしている方は，第3章「**全部まとめて——コンパイルから実行まで**」と，Turbo C 統合デバッガの使い方が示されている第4章，および第12章「**Turbo C での上級プログラミング**」を読む必要があるでしょう。

Turbo Pascal プログラマ

第9章「**Turbo Pascal プログラマへのアドバイス**」は，特に Turbo Pascal プログラマのために用意されています。この章では，Turbo Pascal と Turbo C の対応するプログラムの比較がなされ，2つの言語の間の重要な違いについて詳しく述べられています。

Turbo Pascal でプログラミングの経験を積んでいる方は，プログラミングに関する7つの基本要素（第6章で解説しています）についてはよく理解していると思います。Turbo C を手早く理解するには，まず第5，6，7章を読むとよいでしょう（ボーランドのメニュー形式の製品，SideKick や Turbo Basic などを使用している方は，第5章はざっと目を通すだけでよいでしょう)。また，第3章に目を通して，Turbo C プログラムのコンパイルと実行の手順，Turbo C 統合デバッガの使い方を把握しておくことをお薦めします。

Turbo Prolog プログラマ

すでに Turbo Prolog を使っていて，Prolog のモジュールを Turbo C と結び付けたいという方は，第11章を読んでください。

第2章

さあ始めよう

Turbo Cのシステムへのインストールが済めば，もうすでにプログラミングを始める準備は整っています。しかしその前に，Turbo Cをどうやって走らせるか，プログラムを書いたり直したるするためにエディタをどうやって使うか，またプログラムをどうやってコンパイルし実行するかといったいくつかの基本的な点を理解しておかなければなりません。

好みのASCIIテキストエディタを使ってプログラムを書き，コマンドライン版コンパイラ（Turbo CのTCCバージョン）を使ってDOSのコマンドラインからこれをコンパイルし実行させることができます。しかし，Turbo C統合開発環境（Turbo CのTCバージョン）の方が，少なくとも最初は，こうした作業をより簡単に行なえるということがわかるでしょう。統合開発環境は，エディタ，Turbo Cコマンドのメニューシステム，統合デバッガ，およびTC環境の中からコンパイル・実行が行なえる組み込みのプロジェクトMake機能を提供してくれます。

注意：この章では，練習のステップを進めるのにしたがって，適宜TCのメニューの使い方を説明していきます。TCのメニューシステム全体に関する包括的な解説は第5章にあります。

この章では……

まず最初に，Turbo Cを使う上で必要ないくつかの基本操作を説明します。Turbo C統合開発環境（TC）のロード，Turbo Cへのプログラムのロード，プログラムの作成（コンパイル）と実行の3つです。

次に，TCエディタを使ってどのようにプログラムを修正するかを示します。

最後に，まったく新しいTurbo Cプログラムを書いて，これをコンパイル・実行する前にセーブする手順を示します。

HELLO.C：ファイル1つのプログラムの作成と実行

では簡単に始めましょう。あわてて飛び込んで自分のTurbo Cプログラムを書き始める前に，すでにあるプログラムを使ってTurbo C統合環境バージョン（TC）の練習をしていくことにしましょう。

例題プログラムをインストールしたディレクトリには，たいへんシンプルなソースコードを持つHELLO.Cというファイルがあります。これを使って，単一ファイルのTurbo Cプログラムの作成と実行を行なう6つのステップを見ていくことにします。

ステップ1：Turbo Cを起動する

INSTALLプログラムによってTurbo Cのインストールが済んでいれば，TCはすでにシステムのメインTurbo Cディレクトリにあるはずです。そのディレクトリに入って，MS-DOSのコマンドラインからTCとタイプして**リターン**を押してください。

注意：TCがあるディレクトリではなく，別の作業ディレクトリでプログラミングを行ないたい場合は，TCプログラムがどこにあるかをMS-DOSに知らせなければなりません。

■ TCが入っているディレクトリをMS-DOSのPATHコマンドを使って指定します（MS-DOSユーザーズリファレンスのPATHの節を見てください。TCのパスを設定するときには，PATHにすでに設定されている内容をこわさないように注意してください）。

■ MS-DOS 3.xを使用している場合は，コマンドラインでTCのディレクトリへのパス名をつけます（**例**：¥TURBOC¥TC）。

ステップ2：作業ディレクトリを選択する

カレントディレクトリがHELLO.Cの入っているディレクトリの場合には，このステップは飛ばしてもかまいません。

ロードしたいファイルHELLO.Cが入っているドライブとディレクトリを指定します。これには，まずFileのプルダウンメニューに移動します（***F10***を押してから***F***を押すか，あるいは***GRPH-R***を押してください。IBM PCでは***GRPH***は***Alt***になります）。メニュー

の中から Change dir を選択してください(矢印キーを使ってハイライトバーを動かして**リターン**を押すか，単に ***C*** をタイプしてください)。New Directory というプロンプトボックスが表示されたら，HELLO.C が入っているディレクトリの名前をタイプして**リターン**を押します。このディレクトリがカレントディレクトリになります。

注意：New Directory の入力ボックスには，最初はカレントディレクトリが表示されます。したがって，現在どのディレクトリにいるかを知りたいときには **File/Change dir** を使うことができます。これを選択して New　Directory ボックスの内容を確認したあと，***ESC*** を押せばディレクトリを変更せずにメニューに戻ることができます。

ステップ3：作業環境を設定する

INSTALL プログラムによってインストールを行なっていれば，作業環境はすでにセットアップされています。このステップを飛ばしたくない方は，環境が正しく設定されているかどうかを確認する意味で読んでください。

作業環境を設定するには，***F10*** に続いて ***O*** を押して（あるいは ***GRPH-O***，または ***Alt-O*** を押す)，メインメニューバーから Options メニューを呼び出します。その中から Directories を選択して Directories メニューを表示させます。このメニューでは，Include directories と Library directories の2つの項目が必要です。

Include directories を選んで，Turbo C の標準のインクルードファイル（.h ファイル)が置かれているドライブとディレクトリの名前をタイプしてリターンを押します。ディレクトリを複数指定する場合はセミコロン（;）で区切ります。インクルードディレクトリは通常 A：¥TURBOC¥INCLUDE と A：¥TURBOC¥INCLUDE¥SYS の2つです（IBM　PC では C：＼TURBOC＼INCLUDE と C：＼TURBOC＼INCLUDE＼SYS になります)。この場合は次のようにタイプして**リターン**を押します。

```
A:¥TURBOC¥INCLUDE;C:¥TURBOC¥INCLUDE¥SYS
```

次に Library directories を選んで，Turbo C のライブラリとスタートアップファイルが置かれているドライブとディレクトリの名前をタイプしてリターンを押します。これは通常 A：¥TURBOC¥LIB です (IBM PC では C：＼TURBOC＼LIB)。セミコロンで区切れば，他のディレクトリも同時に指定することができます。

注意：さらに，このメニューの中の Output directory も同じようにして設定して，出力ディレクトリ（コンパイルされたプログラムが格納される場所）を設定することができます。出力ディレクトリを設定すると，以降コンパイラとリンカの出力は，カレントディレクト

りではなく，すべてここで指定したディレクトリに書き込まれます。差し当たってここでは設定する必要はありません。

最も簡単な場合についてですが，これでCのプログラムの作成に必要な設定は済みました。

この作業環境の設定をコンフィギュレーションファイルにセーブしておくと，Turbo Cを起動したときには自動的にその設定内容がロードされるようになります。***ESC***を押してOptionsメニューに戻り，**S**ave optionsを選択してください。Config Fileボックスが現われてデフォルトのファイルTCCONFIG.TCが表示されます。**リターン**を押せば，TCCONFIG.TCがカレントディレクトリに書き込まれます。

注意：Turbo Cを起動すると，TCはカレントディレクトリにTCCONFIG.TCというファイルがあるかどうか調べ，あればそれをロードします。Save optionsのConfig Fileボックスで新しい名前を入力することによって，コンフィギュレーションファイルに別の名前をつけることもできます。ただし，次にTCを起動したときにこのコンフィギュレーションファイルをロードさせるには，TCを呼び出すコマンドラインで/cスイッチを使うか，TCに入ってから**O**ptions/**R**etrieve optionsによって，その名前をはっきり示してやらなければなりません（/cスイッチについては第5章第I部の「TC **コマンドラインスイッチ**」の節を参照してください）。

注意：ある特定のプログラムを開発している際に，デフォルトのコンフィギュレーションファイルをそのプログラムと同じディレクトリに置き，そのディレクトリからTCを呼び出すようにすると便利です。コンフィギュレーションファイルがカレントディレクトリにない場合には，TCはTurbo Cディレクトリを探すようになっています。汎用目的のコンフィギュレーションファイルをTurbo Cディレクトリに1つ置いておき，開発するプログラムに必要な設定を書き込んだコンフィギュレーションファイルを各々のソースファイルと同じディレクトリに入れておくことができます。

ステップ4：ソースファイルをエディタにロードする

ではHELLO.Cをロードしましょう。**F**ileメニューから**L**oadコマンドを選択するか，***F3***を押してください（***F3***はファイルをロードするための**ホットキー**——短縮コマンドです）。Load File Nameボックスが現われ，ワイルドカード指定*.Cが表示されます。HELLOとタイプして**リターン**を押してください。

注意：ロードしたいファイルの名前が不明確なときや，カレントディレクトリにあるすべてのソースファイルの一覧が見たいときには，ファイル名をタイプせずに単に**リターン**を押してください。TC はディレクトリ内にあるすべての.C ソースファイルからなるメニューを表示します。ファイルを選択するには，矢印キーを使ってハイライトバーをロードしたいファイル名に動かして**リターン**を押します。

HELLO.C ファイルがロードされて，TC の Edit ウィンドウに表示されます。内容は次のようになっているはずです。

```
/*      HELLO.C -- Hello, world */

#include <stdio.h>

main()
{
printf("Hello, world¥n");
}
```

注意：TC を起動するときに，ソースファイルとコンフィギュレーションファイルの両方をコマンドラインでの指定によってロードすることができます。これを使うとステップ2から4の作業を省くことができます。統合開発環境はこれを可能にするために，2つのコマンドライン引数を受け取ります。エディタにロードされるソースファイルの名前と，/c スイッチとそのすぐ後に続くコンフィギュレーションファイルの名前です。2つの引数はどちらが先でもかまいません。たとえば次のようにすると，

```
tc hello /cmyconfig
```

HELLO.C がエディタに読み込まれ，コンフィギュレーションファイルとして MYCONFIG.TC がロードされます（/c スイッチとファイル名の間にスペースを入れてはいけないこと，ソースファイルのデフォルトの拡張子は.C，コンフィギュレーションファイルのデフォルトの拡張子は.TC であることをおぼえておいてください)。

ステップ5：実行可能ファイルを作成する

プログラムを作成するときには，まずソースコードをコンパイルしてオブジェクトファイル（拡張子.OBJ を持つマシンコードファイル）を作成します。次にオブジェクトファイルをリンカに渡して，拡張子.EXE を持つ実行可能ファイルに変換させます。リンカは，実行可能ファイルの中に，標準実行時ライブラリファイルから必要なサブルーチンをコピーします（Turbo C には，こうしたライブラリファイルがどこにあるかはすでに伝えてあります。作業環境を設定したステップを思い出してください)。

しかし別の方法もあります。プログラムを作成する一番簡単な方法は，***F10*** を押してから ***C*** を押して（あるいは ***GRPH-C*** を押して）Compile メニューを呼び出し，Make EXE file を選択することです（あるいは Make EXE file のホットキー ***F9*** を押します）。Compile メニューに表示されているコンパイル後のオブジェクトファイルの名前とリンク後の.EXE ファイルの名前をよく見てください。

Make EXE file を選択すると，画面には Compiling ウィンドウが現われ，Turbo C がちゃんとプログラムをコンパイルしリンクするはずです。すべてうまくいけば，Compiling ウィンドウの一番下に Success と表示され，Press any key というメッセージが点滅します。

注意：プログラムにどこか悪いところがあると，Compiling ウィンドウの最下行に Errors（エラー）あるいは Warnings（警告）と表示され，やはり Press any key というメッセージが点滅します。ここでどれかキーを押すと，画面の下の方にある Message ウィンドウにエラーメッセージあるいは警告メッセージが表示されます。こうなってしまった場合には，***F6*** を押して Edit ウィンドウに戻り，プログラムを前の HELLO.C のリストのように正しく修正してからもう一度コンパイルしてください。

ステップ6：プログラムを実行する

この時点で，実行可能プログラムはすでにできているはずです。プログラムを走らせるには，Run メニューから Run を選ぶか，***CTRL-F9*** を押します。

なにが起こりましたか? 画面が1回点滅し，すぐに TC の画面に戻ってしまったはずです。プログラムの出力を見るには，Run/User screen を選ぶか，***GRPH-F5*** を押します。User screen を選ぶと，プログラムが行なった画面出力が表示されているユーザ画面に切り換わります。

ユーザ画面には次のメッセージが表示されているはずです。

```
Hello, world
```

プログラムの出力を調べたら，どれかキーを押して TC の画面に戻ってください。

なにを成し遂げたのか？

それでは，いったん Turbo C の世界から出てください。File メニューから Quit コマンドを選択するか，***GRPH-X*** を押してください。

なにを作り出したのかを見てみましょう。

MS-DOS のプロンプトから dir hello.* とタイプして**リターン**を押してください。次のようなファイルの一覧が表示されるでしょう。

```
HELLO    C       104  88-12-01   2:00
HELLO    OBJ     457  88-12-01  13:16
HELLO    EXE    9486  88-12-01  13:16
```

最初のファイル HELLO.C はプログラムの**ソース**です。これにはプログラムのテキスト（**ソースコード**）が収められています。この内容は，MS-DOS のプロンプトから type hello.c とタイプすれば，画面に表示させることができます。見てわかるように，これはあまり大きなファイルではありません——わずか104バイトです。

2番目のファイル HELLO.OBJ は**オブジェクトファイル**です。これには，Turbo C コンパイラが生成したバイナリのマシン語命令（**オブジェクトコード**）です。MS-DOS の TYPE コマンドで内容を見ようとすると，わけのわからないものが表示されるでしょう。

最後のファイル HELLO.EXE が，Turbo C のリンカが作成した実際の**実行可能プログラム**です。これには，HELLO.OBJ のコードだけでなく，リンカがライブラリファイルからコピーした必要なサポートルーチン（**printf** など）がすべて含まれています。実行可能ファイルを走らせるには，MS-DOS のプロンプトから，その名前を，拡張子.EXE はつけずにタイプします。

プロンプトに対して，hello とタイプして**リターン**キーを押し，HELLO.EXE を走らせてみてください。Hello, World というメッセージが画面に表示され，ふたたび MS-DOS のプロンプトが表示されるでしょう。

プログラムを編集する

伝統的に，最初に C で書くプログラムは，カーニハン&リッチィの‘古典的’名著「**プログラミング言語 C**」にある Hello,world プログラムということになっています。これが，いま作成と実行をやり終えたばかりの小さなプログラム HELLO.C です。

では Turbo C 統合開発環境にいくらか慣れてきたところで，すこし自分でプログラミングをしていくことにしましょう。まず HELLO.C をちょっと手直しすることから始めます。それには TC エディタの使い方をおぼえなければなりません。

まだ TC エディタに戻っていなければ，DOS のプロンプトから tc hello とタイプして Turbo C に入ってください。Turbo C に戻って先程のプログラムがロードされるでしょう。

こんどは，ちょっと対話ができるようにプログラムを手直しすることにします。

画面の左上隅，プログラムの最初の行の左端で点滅しているカーソルに注目してください。矢印キーによってこのカーソルを Edit ウィンドウの中で動き回らせることができます。コードを入力するには，入力したい位置にカーソルを動かしてコードをタイプします。***CTRL-Y*** を押すと1行の削除, ***CTRL-N*** を押すと1行の挿入が行なえます。エディタが挿入モードになっていることを確認してください(挿入モードのときには Edit ウィンドウの一番上にあるステータス行に Insert という単語が表示されています。そうなっていない場合は ***INS*** キーを押して切り換えてください)。なお，TC エディタの使い方に関する完全な解説については，第5章および付録 A を参照してください。

では，プログラムを以下のように編集してください。

```
/*      HELLO.C -- Hello, world */

#include <stdio.h>

main()
{
    char    name[150];

    printf("What's your name ?¥n");
    scanf("%s",name);
    printf("Hello, %s¥n", name);
}
```

HELLO.C に3行を付け加えました。第1行 (char name[150];) は，*name* という名前の変数を定義しており，この変数には最大150個までの文字（文字，数字，記号など）を入れることができます（150番目の位置は，後で説明する特殊な文字のために予約されています)。追加した2番目の行は，関数 **printf** を呼び出して What's your name ? というメッセージを書き出します。3番目の新しい行は，関数 **scanf** を呼び出して変数 name に名前を読みこみます。

では次に，***CTRL-F9*** を押してプログラムを走らせてください。Turbo C にはソースコードが変更されたことがわかっているので，プログラムを実行する前に再コンパイルすることに注意してください。

今度はプログラムを走らせると，2つのことが起こります。まずユーザ画面が切り換わって What's your name ? というメッセージが表示され，カーソルは次の行に来て入力待ちになります。あなたの名前をタイプして**リターン**を押してください。次は ***GRPH-F5*** を押します。今度はユーザ画面に Hello, 〈あなたの名前〉 と表示されます。名字と名前の両方を（空白で区切って）入力した場合には，先に入力した方だけしか表示されなかったでしょう。どうしてこうなったかは第6章で学ぶことになります。ここではどれかキーを押して TC 画面に戻ってください。

間違えてしまったときには

プログラムを作っていると，エラーを出してしまったり，警告を受けたりすることがあります。**エラー**というのは，Turbo C がそれをコンパイルしてオブジェクトコードの生成ができなくなるような，プログラム上の誤りのことです。**警告**は，明らかなエラーではありませんが，問題を引き起こしかねない点を指摘するメッセージのことです。エラーや警告があると，TC 画面の下の方にある Message ウィンドウにそのリストが表示されます。エラーと警告は非常にたくさんあり，リファレンスガイドの付録 A でそのすべてを解説しています。

出力をプリンタに送る

HELLO.EXE プログラムの出力を，画面ではなくてプリンタに送るにはどうすればよいかと考えている方もいるでしょう。ここでそのやり方をお教えしますが，プログラムがどのように動くかについては立ち入らないことにします。この段階については，もう充分理解できたはずなので，楽しみはあとにとっておくことにしましょう。

HELLO.C をエディタにロードして，以下のように修正してください。

```
/*     HELLO.C -- Hello, world */

#include <stdio.h>

main()
{
    fprintf(stdprn,"Hello, world¥n");
}
```

プリンタが印字可能な状態になっていることを確認して，前にやったのと同じように ***CTRL-F9*** を押して，プログラムをコンパイル・実行してください。プリンタは，Hello,world というメッセージをプリントアウトするはずです。

ここでは，**printf** のかわりに **fprintf** を使用しました。Turbo C の経験を積んでリファレンスガイドにも踏みこんでいくようになれば，ここで追加したような要素についても理解が深まっていくことと思います。

Turbo Cの2番目のプログラムを書く

こんどはHELLO.Cプログラムをもう少し修正して，それを新しいファイルに格納してみましょう。まだエディタの中にいるはずですが，そうでない（点滅するカーソルがどこにもない）場合には，ホットキー ***GRPH-E*** を押すか，***F10*** を押してメニューシステムをよびだしてから ***E*** を押して Edit を選択してください。以下のようにプログラムを変更してください。

```
#include <stdio.h>

main()
{
    int  a,b,sum;

    printf("Enter two numbers:  ");
    scanf("%d %d",&a,&b);
    sum = a + b;
    printf("The sum is %d ¥n",sum);
}
```

ここでは元のプログラムに5ヶ所の変更をくわえました。

- *name* を定義していた行を別の3つの変数の定義に置き換えました（*a*，*b*，*sum*，すべて整数型）。
- **printf** 文の中のメッセージを変更しました。
- **scanf** 文の中の，書式文字列（"%d %d"）と変数リスト（&a,&b）を変更しました。
- sum = a + b; という代入文を追加しました。
- 最後の **printf** 文の中の書式文字列と変数リストを変更しました。

パーセント記号（%），アンパサンド（&），円記号（¥）などに惑わされないようにしてください。こうした記号がどんな意味を持っているかは第6章で説明します。

ディスクに書く

今はまだ，***F2*** を押さないでください。押してしまうと，このプログラムはHELLO.Cとしてセーブされてしまいます（ここではちがう名前でプログラムをセーブすることになっ

ていますから)。

F2 のかわりに ***GRPH-F*** を押して File メニューに入ってください。次に ***W*** を押して Write to コマンドを呼び出します。Turbo C は，このプログラムにつける新しい名前を入力するよう求めてくるので，sum.c とタイプして**リターン**を押してください。これで，2番目のプログラムが SUM.C としてディスクにセーブされました。

SUM.C を走らせる

CTRL-F9 を押してください。Turbo C はプログラムをコンパイルします。なにかエラーが出たら，エディタに戻って，プログラムが正しく入力されているかどうか確認してください。

エラーが出なければ，Turbo C は適切なライブラリルーチンをリンクして，プログラムを実行します。ユーザ画面に切り換わって，次のメッセージが表示されるでしょう。

```
Enter two values:
```

プログラムは，あなたが2つの整数値を，**空白**，**タブ**，または**リターン**で区切って入力するのを待っています。2番目の値の後に**リターン**を押すのを忘れないでください。今度はプログラムは，その2つの値の和をユーザ画面に書きます。**Run/User screen** を選択して(あるいは ***GRPH-F5*** を押して) 結果を見てください。なにかキーを押せば Turbo C に戻ります。

おめでとうございます! プログラミングの基本的な要素をいくつか使って，Turbo C のまったく異なる2つのプログラムを書くことができました。基本的な要素ってなんだ，とお思いでしょう。これらについても第6章を読めばわかってくるでしょう。

第3章

全部まとめて——コンパイルから実行まで

前の章でいくらか Turbo　C を使う経験を積んだので，こんどは少しこみいった問題 Turbo C 統合開発環境の上級機能とコマンドライン版の TurboC に話を進めていきましょう。

Turbo C は，C のプログラム開発に適した非常にフレキシブルな環境を与えてくれます。マスターディスクはすぐにでも動かせるようにデフォルトの設定がなされていますが，この設定はユーザの必要に応じて簡単に変更できるようになっています。さらにさまざまなサポートツールが用意されており，プログラム開発に必要となる，エラー原因の追跡やファイルシステムの管理といった作業が容易に行なえるようになっています。

ボーランドの使いやすい統合開発環境（TC）にまだあまり慣れていない方は，TC のメニューシステムでプログラムをコンパイル・実行する前に，第5章に目を通しておいてください。順序立ててやさしく説明されており，TC を快適に使えるようになるのにさほど時間はかからないでしょう。

この章では……

Turbo C は，統合開発環境からでも MS-DOS のコマンドラインからでもプログラムのコンパイルと実行ができるので，この章ではその両方について説明します。しかしながら，統合開発環境の方がより完全なパッケージであり，強力かつ使いやすいので，そちらから先に紹介したほうがよいでしょう。

この章では，まず Turbo C プログラムを統合開発環境でコンパイル・リンクして実行可能ファイルを作成するまでの手順を復習し，そのあとで TC のデバッグ機能についてその概要を見ていきます。

次に，統合開発環境の中からどうやってプログラムを走らせるかを説明し，また TC に組み込まれているプロジェクト機能——プロジェクト Make とその使い方について示します。

統合開発環境からのプログラムの実行方法を説明したあと，コマンドライン版を使ってどうやって Turbo C プログラムをコンパイル，リンク，Make(再作成)，実行するかを説明します。Turbo C には，統合開発環境版に加えて，スタンドアロンのコンパイラ（TCC)，リンカ，MAKE ユーティリティが含まれています。これらの詳細については付録 B と付録 C で詳しく解説しています。

再び, TC でのプログラムの作成

Turbo C の統合開発環境 (TC) で新しいプログラムを作成するときには，通常は以下のような手順を踏みます。

1. 環境に関するいくつかのオプションを設定して，コンパイラとリンカがさまざまなファイルの探索や出力を行なうディレクトリを決めておきます。
2. 作成しようとするプログラムを TC エディタにロードします (**注意**：プログラムが複数のモジュールから構成される場合には，モジュール名を並べた**プロジェクトファイル**を作ることになります)。
3. 実行可能なプログラムファイルを作成します。

この手順は実際には，ソースプログラムが1つのファイルで構成されるか，複数のファイルで構成されるかによって異なってきます。

プログラムのデバッグ

プログラムのエラーを見つけてそれを修正していくのは，プログラミングでは煩わしい作業の一つです。Turbo C 統合開発環境（TC）は，コンパイラ時・実行時の両方において，プログラマの助けとなるデバッグ機能を提供しており，その作業を非常にたやすいものにしてくれます。

構文エラーをとらえる：エラー追跡機能

TC を使う大きな理由の1つは，構文（コンパイル時）エラーの修正がたやすく，またコンパイラが出す警告メッセージの評価に手間がかからないことです。TC は，コンパイラとリンカが出力するメッセージをバッファにおさめ，これらを Message ウィンドウに表示します。この機能によって，すべてのメッセージを眺めながら，ソースファイルの修正を進めることができるのです。

これを試してみるために，HELLO.C プログラムに故意に構文エラーを作ってみましょう。まず，3行目のインクルード文から先頭の#を取り除きます。さらに，7行目の **printf** のカッコ( )内の文字列の最後の二重引用符"も削除します。バグを入れたプログラムは次のようになるでしょう。

```
/*      HELLO.C -- Hello, world */

include <stdio.h>

main()
{
    printf("Hello, world¥n);
}
```

GRPH-F9（Compile to OBJ を呼び出すホットキー）を押して，ファイルをもう一度コンパイルしてください。Compiling ウィンドウには，エラーと警告（Warning）の数が表示されます（ここでは，エラーが2個で警告はないはずです）。

Message ウィンドウ

Compiling ウィンドウに Press any key と表示されたら**スペースバー**を押してください。Message ウィンドウがアクティブになり，ハイライトバーが最初のエラーまたは警告に位

置づけられます。その最初のエラーは現在エディタにロードされているファイルの中で起こったので，エディタの中にもハイライト表示された行が1行あるはずです。これは，ソースコードの中でコンパイラがエラーまたは警告を出した場所を指し示しています。

ここで上下の矢印キー（↑↓）を使うと，Messageウィンドウ内のハイライトバーを移動させて，他のメッセージを確認することができます。こうすると，メッセージの移動に伴って，Editウィンドウ内のハイライトバーもそのエラーが起きた場所を順に示していきます。またハイライトバーをCompiling ...に移動させると，エディタはそのファイル内での直前の位置（ここでは ***GRPH-F9*** を押したときにカーソルのあった場所）を示します。

Messageウィンドウのメッセージが長すぎて全部が見えない場合には，左右の矢印キー（←→）を使ってウィンドウを横スクロールさせることができます。より多くのメッセージを同時に見たければ，***F5*** を押してウィンドウを拡大することができます。Messageウィンドウを拡大するとEditウィンドウは見えなくなり，エラーの追跡は行なえなくなるので，ここではウィンドウを分割画面モードのままにしておいてください。

構文エラーを修正する

エラーを修正するには，まずMessageウィンドウのハイライトバーを最初のメッセージに置いてから**リターン**を押します。カーソルがEditウィンドウに移り，エラーの原因となった箇所に位置します。エディタのステータス行には選択したエラーメッセージが表示されていることに注目してください（これはウィンドウを拡大しているときに便利です）。ここでメッセージに対するエラーを修正することができます（最初に取り除いた#を元の位置に書き込んでください）。

エラーメッセージが複数ある場合，次のエラーを修正するには2つの方法があります。

1つは，***F6*** を押してMessageウィンドウに戻り，先程と同じやり方で，修正したいメッセージを選択する方法です。

しかし次のエラーに移るのにわざわざMessageウィンドウに戻る必要はありません。単に ***GRPH-F8*** を押せば，エディタは，Messageウィンドウの次のエラーに対応する箇所にカーソルを移動させます。また，***GRPH-F7*** を押せば，1つ前のエラーに戻ることができます。

2つの方法にはそれぞれ利点があり，どちらがよいかは状況によって異なってきます。ちょっとしたタイプミスなどの些細な間違いによって，エラーメッセージがたくさん出てしまうことがよくあります。こんなときには，最初のエラーを選択して修正すれば，その後のエラーのいくつかは意味のないものになってしまうでしょう。このような場合には1番目の方法で，最初のエラーを修正したらMessageウィンドウに戻ってエラーをいくつか飛び越し，意味のある次のエラーを選択すればよいわけです。そうでない場合には，2番目の方

法で，エラーを1つ修正したら **GRPH-F8** を押して次のエラーに移るようにすればよいでしょう。

GRPH-F7 と **GRPH-F8** はホットキーであるということを憶えておいてください。つまりTCの中であればどこからでも使えるキーです。したがって，Messageウィンドウにいるときに **GRPH-F8** を押すと，現在ハイライトされているメッセージではなく，その次のメッセージを選択することになります（現在のメッセージを選択するにはリターンを押します）。その後にメッセージがなければ **GRPH-F8** を押しても何も起こりません。

注意：リンカが出したメッセージは上で示した方法では選択できず，ソースファイルの中を追跡することはできません。

構文エラーを修正していく中では，テキストを追加したり削除したりする必要が出てきます。このような変更が行なわれてもエディタはそれを認識しており，次のエラーは正しく位置づけられるようになっています。プログラマがエラーの起こった行番号や，追加や削除した行数を憶えておく必要はありません。

実行時エラーをとらえる：統合デバッガ

構文エラーをすべて修正すれば，プログラムはうまくコンパイルできるはずです。しかし，そのプログラムが**ロジック**（実行時）エラーを含んでいれば，期待通りには動かないこともあります。こんどはエラー追跡機能はなんの助けにもなってくれません。

実行時エラーをとらえるために，TCは**統合デバッガ**を備えています。統合デバッガを通じてプログラムを実行すると，好きな場所で実行を止めたり，変数の値をチェックしたりすることができ，さらに値を変更してプログラムがどのように反応するかをテストすることもできます。TC統合デバッガの使い方については，第4章で練習することになっています。

プロジェクト：複数のソースファイルを扱う

TCの優れた機能の1つに，複数のソースファイルからなるプログラムの分割コンパイル機能があります。TCの**プロジェクト Make** 機能は，これをたいへん効率的に行ないます。

第2章の例題ではソースファイルは1つだけだったので，Compile/Make EXE file コマンドを選ぶだけで実行可能ファイルを作成することができました。しかし，複数のCソースファイルからなるプログラムを作成する場合には，どんなファイルで構成されるのかをTCに伝えてやる必要があります。このためには**プロジェクトファイル**を作らなければなりません。

プロジェクトファイルを作るのは，Cソースファイルの名前をただ並べて書くだけという簡単な作業です。プロジェクトファイルには種類の違うたくさんのファイルを書けるということもいずれわかってくるでしょうが，ここでは，ファイル2つのプログラムというごく簡単な例から始めましょう。

1つの基本的なケースとして，メインプログラムのファイルと，メインプログラムが参照する関数やデータが定義されているサポートファイルの2つ，という場合があります。たとえばメインプログラム MYMAIN.C は以下のようなものだとします。

```
#include <stdio.h>

char *GetString(void);

main (int argc, char *argv[])
{
    char *s;

    if (argc > 1)
        s = argv[1];
    else
        s = "the universe";

    printf("%s %s.¥n",GetString(),s);
}
```

また，サポートファイル MYFUNCS.C は次のようになっているとします。

```
char ss [] = "The restaurant at the end

char *GetString(void)
{
    return ss;
}
```

では，MYMAIN.C と MYFUNCS.C を作ってください。

次に，この2つのファイルでプロジェクトファイルを作ります。想像された方もいるかと思いますが，プロジェクトファイルには，コンパイルされリンクされるこうしたファイルの名前が並んでいるだけです。新しいファイルを作って，次のように2つのファイル名を入力してください。

```
mymain
myfuncs
```

.C をつける必要はありません。TC は拡張子のないファイルは.C ファイルとみなします（つけたければ.C をつけてもかまいません）。ファイル名の順序は，ただその順でコンパイルされるということ以外には大きな意味はありません。プロジェクトファイルを次のようにしても，上の場合と同じ結果が得られます。

```
myfuncs
mymain
```

このファイルを MYPROG.PRJ としてセーブしてください（**File/Write to** を選択します）。これで終わりです。

プロジェクトファイルの名前（MYPROG.PRJ）は，メインファイルの名前（MYMAIN.C）とは異なっていること注意してください。2つのファイルの名前は同じであっても（拡張子は異なります），異なっていてもかまいません。憶えておいてほしい重要な点は，実行可能ファイル（およびリンカが出力するマップファイル）の名前は，プロジェクトファイル名から決定されるということです。したがってこの場合，実行可能ファイルは MYPROG.EXE（マップファイルは MYPROG.MAP）になります。

また，プロジェクトファイルの中に書き込むファイル名には，パス名を指定することもできます。これによって，すべてのファイルが同一のディレクトリに入っていなくても，プログラムを作成することができます。

複数のソースファイルのプログラムを作成する

プロジェクトファイルはすでにできているので，あとはプロジェクトファイルがどれなのかを TC に知らせるだけです。これには，**Project** メニューからプロジェクトファイルの名前を入力します。***GRPH-P*** を押して **Project** メニューに入り，**Project name** を選択します。プロジェクトファイルの名前をそのまま入力してもかまいませんし，ワイルドカードを使って（*.PRJ など）指定のディレクトリ内のプロジェクトファイルの一覧を表示さ

せ，その中から選んでもかまいません(プロジェクトファイルをまだ1つもセーブしていなければ，何も出てきませんが)。プロジェクト名を入力したら，あとは **F**9 (Make) を押せば実行可能ファイルが作られます。また，このプログラムを走らせるには，**CTRL-F9**(Run/Run) を押します。

プログラムを走らせること(Run)の中には，Make することも含まれていることに注意してください。したがって **CTRL-F9** を押すと，そのプロジェクトが再コンパイルの必要がある場合には，コンパイル→リンクのサイクルが開始されます。したがって，いちいち **F9** を押して Make を行なわなくてもよいわけです。Run/User screen を選択して(あるいは **GRPH-F5** を押して)出力を確認してください。どれかキーを押すと TC エディタに戻ります。

再び，エラー原因を追跡する

ファイル1つのプログラムの例では，警告やエラーメッセージを引き起こした構文エラーは，Message ウィンドウの中で確認し選択することができました。複数のファイルのコンパイル（あるいは Make）の場合でも，Message ウィンドウは同様に使うことができます。

これを見るために，2つのファイル (MYMAIN.C, MYFUNCS.C) に構文エラーをいくつか入れてみましょう。まず MYMAIN.C では，インクルード文の左括弧 (〈) と，8行目の **char** の *c* を取り除いてください。このように変更すると，MYMAIN では3つのエラーと3つの警告が出るようになるはずです。

次に MYFUNCS.C をロードして，5行目の return から先頭の *r* を削除します。こうするとエラーが2つと警告が1つ出るはずです。

このような変更を行なうと，2つのソースファイルの日付(時刻)は，そのオブジェクトファイル (.OBJ) より古くなるので，Make を行なうと再コンパイルが行なわれることになります。複数ファイルの場合のエラー追跡のようすを見るには，プロジェクト Make がどの段階で動作を中止するかという基準を変えてみる必要があります。これは Project メニューの中の特殊なトグルを設定にすることによって行ないます。

Make を停止させる

TC では，Make のサイクルが停止する条件はいくつかあります。実行可能ファイルの作成が終了すれば，もちろんプロジェクト Make は止まりますが，なんらかのエラーを示して停止する場合もあります。

たとえば，プロジェクトファイルの中のソースファイル（あるいは依存ファイル）のどれかが見つからない場合などです（依存ファイルについては後で説明します）。また，**CTRL-STOP**（IBM PC では **Ctrl-Break**）を押せば，プロジェクト Make を強制的に停止させることもできます。

その他にも，コンパイラがメッセージを出力した時点で止まるようにすることもできます。Project メニューの Break make on オプションの設定によって，Make が停止するメッセージの種類を選択することができます。このトグルのデフォルトは Errors（エラー）になっており，通常はこれを使うことになるでしょう。このオプションでは，Make がファイルをコンパイルする際にいつ停止するかを，警告（Warnings）が起きたとき，エラー（Errors）が起きたとき，致命的なエラー(Fatal errors)が起きたとき，あるいは（エラーには関係なく）リンクの前，の中から指定することができます。

どのモードを選べば有効かは，実際にはユーザがどんな方法でエラーや警告を修正していくかによって決定することになります。エラーや警告を見るとすぐに直したくなる人は，Break make on オプションを Warnings（警告）または Errors（一般エラー）にしておくとよいでしょう。エラーをすべて出しきってから修正を始めるという人は，Fatal errors（重大なエラー）または Link（リンク前）にセットしておけばよいでしょう。

ソースファイルが複数の場合の構文エラー

複数ファイルでのエラーを見るために，ここでは Break make on を Fatal errors にセットしてください。これには，**GRPH-P** を押して Project メニューを呼び出して Break make on を選択します。そして Project/Break make on メニューから Fatal errors を選択します。

もうすでに，MYMAIN.C と MYFUNCS.C には構文エラーを組み込んであるはずです。**F9**（Make）を押して，プロジェクトを Make してください。Compiling ウィンドウには，コンパイル中のファイルと，各ファイルのエラーと警告の個数，および Make 全体のエラーと警告の合計数を表示します。Press any key が点滅したらスペースバーを押してください。

カーソルは，Message ウィンドウの最初のエラーまたは警告に位置しているはずです。そして，そのメッセージが参照しているファイルがエディタ内にあれば，Edit ウィンドウのハイライトバーは，コンパイラがその問題を発見した場所を示しています。前と同じように，矢印キーで Message ウィンドウをスクロールさせると，別のメッセージを選択することができます。コンパイルされるファイルごとに Compiling ...というメッセージがあることに注目してください。このメッセージはエラーや警告ではなく，出力されるメッセージをファイルごとに区切るためのものです。

このファイルの区切りを越えてスクロールさた場合，Options/Environment メニューのMessage tracking トグルの設定によって，Edit ウィンドウで次のファイルが追跡されるかどうかが変わってきます。デフォルトでは現在エディタ内にあるファイルの中でのみ追跡が行なわれるようになっています。

したがってここでは(デフォルトのはずなので)，エディタには入っていないファイルに対するメッセージに移ると，Edit ウィンドウ内のハイライトバーは消えてしまいます。エディタ外のファイルに対するメッセージを選択する（**リターン**を押す）と，TC はそのメッセージが参照しているファイルをエディタにロードし，エラーの場所にカーソルを位置づけます。***F6*** (Message) を押して Message ウィンドウに戻ると，追跡は今度はそのファイルの中で行なわれるようになります。

Message tracking トグルを All files にセットしておくと，ファイルの区切りを越えて自由にメッセージを選択することができます。つまり，Message ウィンドウをファイルの区切りを越えてスクロールさせると，TC は該当のファイルを自動的にエディタにロードするので，各メッセージとソースファイルの対応箇所がすぐにわかるわけです。試してみてください。

Message tracking トグルを Off にセットすれば，完全に追跡をやめてしまうこともできます。この場合には，修正したいメッセージを選択して**リターン**を押します。そのメッセージが参照しているファイルがエディタにロードされ，エラーの場所にカーソルが位置づけられます。

GRPH-F7（Previous error）と ***GRPH-F8***（Next error）は，Message tracking トグルの影響は受けません。この2つのホットキーは，いつでも次あるいは前のエラーを探して，必要ならば該当するファイルをエディタにロードします。

メッセージの保持と消去

通常，Make を始めるときには，Message ウィンドウは新しいメッセージを入れるためにクリアされます。しかし，前の Make のメッセージを取っておきたい場合もあります。

次のような場合を考えてみましょう。ソースファイルが何本もあるプロジェクトがあって，Break make on を Errors にセットしているとします。このプロジェクトを Make したときに，いくつかのファイルで警告メッセージが出力され，あるファイルがエラーを起こして Make が停止したとします。そこでエラーを修正して，コンパイラがその修正を受け入れてくれるかどうかを確かめるわけですが，もう一度 Make かコンパイルを行なうと，前回出力された，まだ見る必要のある警告メッセージは消えてしまいます。どうすればこれを避けることができるでしょうか。これには，単に Options/Environment メニューの Keep messages トグルを On にしておけばよいのです。

Keep messages トグルが On になっていると, Make が始まるときに Message ウィンドウはクリアされません。消去されるメッセージもありますが，それは再コンパイルするファイルに関するものだけで，古いメッセージが新しく出力されるメッセージに置き換えられます。

メッセージに対する措置が完了したら，その時点で Project/Remove messages を選択すれば，Message ウィンドウをクリアすることができます。Keep messages を Off にしておくと，Make を行なうたびに古いメッセージはすべて消去されます。

別のプロジェクトへ移るときには，Message ウィンドウをクリアしておくようにするとよいでしょう。Project メニューから Clear project を選択すれば，これを簡単に行なうことができます。Clear project は，プロジェクト名と Message ウィンドウの両方をクリアします。Project/Clear project を選択したあとは，新しいプロジェクトを定義することもでき，ファイル1つのプログラムをエディタにロードしたり，Primary C file コマンドでプライマリファイル名を指定してコンパイル・実行を行なうこともできます。

プロジェクト Make のパワー

プロジェクトの Make の前の説明では，C のソースファイルの名前を並べておくという，もっとも基本的な使い方について述べました。プロジェクト Make には，さらに強力な機能があり，もっと複雑な状況を取り扱うこともできます。これを知るには，まず Make がどのように働くのかを理解する必要があります。

Make は，ソースファイルの作成日時と，コンパイラが作成したオブジェクトファイルの作成日時を比較します。この作成日時の比較によって，ソースファイル名が並んでいるプロジェクトリスト内の**暗黙の依存関係**が定義されます。

前に示した MYPROG.PRJ を例にとると，次のような依存関係が定義されていることになります。

- MYMAIN.OBJ は MYMAIN.C に依存する
- MYFUNCS.OBJ は MYFUNCS.C に依存する
- MYPROG.EXE は MYMAIN.OBJ, MYFUNCS.OBJ および MYPROG.PRJ に依存する

これは，MYMAIN.C の作成日時が MYMAIN.OBJ より新しければ，MYMAIN.OBJ は（当然のことながら）古いということになり，したがって MYMAIN.C は再コンパイルしなければならないということを意味しているわけです。実行可能ファイル（.EXE）は，

プロジェクトに含まれるすべてのオブジェクトファイルに依存するとともに，プロジェクトファイルそのものにも依存しているという点に注意してください。つまり，オブジェクトファイルのいずれか，あるいはプロジェクトファイル（MYPROG.PRJ）がMYPROG.EXEより新しい場合には，Makeは再度リンクを行なってMYPROG.EXEを作成することになります。こうした暗黙の依存関係は，プロジェクト内のCファイルの単純なリストから導き出されます。

明示的な依存関係

もっと大きなプロジェクトになると，依存関係をはっきりと指定できるような，より高度な機能が要求されるようになります。Cのソースファイルが，別のソースファイルに依存しているような場合に，こうした機能が必要です。一般的にCのソースファイルには，外部ルーチンとのインターフェースを定義しているヘッダファイル（.H）をインクルードしています。このようなルーチンとのインターフェースが変更された場合には，そのルーチンを使用しているファイルはすべて再コンパイルしなければなりません。これを行なうのが**明示的な依存関係**です。

たとえば，メインプログラムMYMAIN.CがヘッダファイルMYFUNCS.Hをインクルードしているとします。また，プロジェクトファイルには次のように依存関係が指定されているとします。

```
MYMAIN.C (MYFUNCS.H)
MYFUNCS  (MYFUNCS.H)
```

ここで，MYFUNCS.Hを変更してからMakeを行なうと，MYMAIN.CとMYFUNCS.Cが再コンパイルされます。

上のプロジェクトファイルでは，MYFUNCS.CもMYFUNCS.Hに依存することが示されていることに注目してください。これによってファイルの正しい対応関係が維持されています。したがって，明示的な依存関係のほかに，次のような暗黙の依存関係が示されていることになります。

- MYMAIN.OBJはMYMAIN.CとMYFUNCS.Hに依存する
- MYFUNCS.OBJはMYFUNCS.CとMYFUNCS.Hに依存する
- MYPROG.EXEはMYMAIN.OBJ，MYFUNCS.OBJ，およびMYPROG.PRJに依存する

プロジェクトファイルでは，各Cファイルの明示的な依存関係は，必要な数だけ書くこ

とができます。Cファイルが依存するファイルは，空白，カンマ(,)，またはセミコロン(;)で区切って，カッコの中に入れておけばよいのです。

たとえばMYMAIN.Cが，MYFUNCS.H，YOURS.H，およびOTHER.Hに依存するのであれば，次のように書いておきます。

```
MYMAIN  (MYFUNCS.H, YOURS.H, OTHER.H)
```

注意：自動依存チェック（Project/Auto dependencies）がオンの場合には，インクルードファイルはすべてチェックされますが，明示的な依存関係はチェックされません。

自動依存チェック

プロジェクトMakeには，プロジェクトリスト内のソースファイルとそれに対応するオブジェクトファイルの依存関係を自動的にチェックするという能力もあります。プロジェクトMakeは.OBJファイルをオープンして，その.OBJファイルのソースコードが含まれているファイルに関する情報を探します。この情報は，TCまたはTCCがソースモジュールをコンパイルするときに必らず.OBJファイルの中に書き込みます。そして，.OBJファイルを作成するために使用されたすべてのファイルの日付・時刻がチェックされ，.OBJファイルの日付・時刻情報と比較されます。.OBJファイルが，それを構成したファイルよりも古ければ，再コンパイルが行なわれることになります。

注意：この機能を働かせるには，統合環境の中でProject/Auto dependenciesスイッチをOnにしておかなければなりません。

依存関係についてはこれだけです。この方法で以前からあるMakeと同様な機能が得られますが，むずかしい文法などはありません。

Makeの他の機能

Makeのパワーに加えてさらに2つの特徴があります。

- ■リンクする外部オブジェクトファイルやライブラリファイルをプロジェクトの中で指定することができます。
- ■標準のスタートアップルーチンやライブラリを無効にすることができます。

外部オブジェクトファイルとライブラリファイル

アセンブラなど他の言語や，他のコンパイラで作成したルーチンを利用したいという場合がときどきあります。また，標準のライブラリにはない特別な用途のライブラリを使いたいこともあるでしょう。こうした場合には，プロジェクトファイルの中に，そのオブジェクトファイルやライブラリファイルの名前を，拡張子をつけて書き入れておきます。たとえば次のようになります（ファイルを並べる順序は重要ではありません）。

```
MYMAIN   (MYFUNCS.H)
MYFUNCS  (MYFUNCS.H)
SPECIAL.OBJ
OTHER.LIB
```

プロジェクト Make は，拡張子.OBJ のついたファイルがあると，そのファイル名をリンクするファイルのリストに含めます。そのファイルをコンパイルしたり，そのソースファイルを探したり，また依存関係情報を読み出そうとしたりはしません。同様に，拡張子.LIB のついたファイルがあると，外部参照の解決のためにリンカが探索するライブラリのリストの中に，そのファイル名を含めます。やはり，コンパイルしたりライブラリを作成したりはしません。

このようなタイプのファイルについては，明示的な依存関係のリストを指定することはできません(指定しても無視されます)。ただし，逆に C のソースファイルの依存関係のリストに，こうしたファイルの名前を含めることはできます。例をあげましょう。

```
MYMAIN   (MYFUNCS.H, SPECIAL.OBJ)
MYFUNCS  (MYFUNCS.H, OTHER.LIB)
SPECIAL.OBJ
OTHER.LIB
```

こうすると，なんらかの理由で.OBJ や.LIB が更新されたときには，C のソースファイルは再コンパイルされることになります。

標準ファイルを無効にする

場合によっては，標準のスタートアップルーチンあるいはライブラリを無効にする（リンクしない）必要が出てくるでしょう。これは，上級ユーザにしか薦められない機能で，初心者にとっては一般的な使い方ではありませんが，必要だと感じたら，ここで説明する方法を参考にしてください。

スタートアップファイルを無効にするには，プロジェクトファイル内の最初のファイル名として C0*x*.OBJ を置かなくてはなりません。*x* には，なんらかの DOS のファイル名を指定します（C0MINE.OBJ など）。重要な点は，名前が C0（シー，ゼロ）で始まっている

こと，プロジェクトファイル内の先頭に指定すること，ファイルの拡張子が.OBJであること，の3つです。

標準ライブラリを無効にするには，プロジェクトファイル内のどこかに，標準ライブラリのかわりに使用する特別なライブラリファイルの名前を書き込むだけです。ライブラリファイルの名前は，Cで始まり，2文字目がメモリモデルを表わす英字(たとえばスモールモデルならS)でなければなりません。残りの最大6個までの文字にはなにをつけてもかまいません。また拡張子は.LIBでなければなりません（たとえばCSMYFILE.LIB, CSNEW.LIBなどです)。

標準ライブラリが無効になっている場合，Makeは標準の数学ライブラリをリンクしようとはしません(無効でない場合には，数学ライブラリは**O/C/Code generation**メニューの**Floating point**トグルにしたがってリンクされます)。標準ライブラリを無効に指定している場合に，数学ライブラリなどをリンクしたいのであれば，プロジェクトファイルの中でそのようなファイルを明示的に指定しなければなりません。

コマンドラインからのコンパイルとリンク

Turbo Cのプログラムは，統合開発環境に加えて“古風な”コマンドライン版でも動かすことができます。プログラムの開発と実行には統合環境がベストです（と思われます）が，状況によってはコマンドラインを使いたいことがあるかもしれません。ある種の高度なプログラムでは，コマンドライン版でなければ処理しきれないということもあります。たとえば，Cのプログラムにインラインのアセンブリコードが含まれている場合などは，統合開発環境版ではなく，コマンドライン版のTurbo C（TCC）を使う必要があります。

TCCは，Cのソースファイルをコンパイルしてリンクし，実行可能ファイルを作成します。TCCは，UNIXのccコマンドと似たような働きをします。TCCは，TASMを呼び出して.ASMファイルをアセンブルすることもできます。コンパイル(およびアセンブル)だけを行ないたい（リンクしない）場合には，コマンドラインで-cオプションを指定します。

TCCのコマンドライン

コマンドラインからTurbo Cを呼び出すには，MS-DOSのプロンプトに対してTCCと入力し，そのあとに必要な引数をつづけます。コマンドライン引数には，コンパイラオプション，リンカオプション，およびファイル名があります。コマンドラインの一般的な書式は次のようになります。

```
tcc [オプション オプション ...] ファイル名 ファイル名 ...
```

コマンドライン上のオプション

コマンドラインのオプションは，頭にダッシュ（-）をつけ，tccコマンドや他のオプションおよびそのあとのファイル名とは，最低1つの空白で区切らなければなりません。オプションのあとにさらにマイナス記号をつけると，そのオプションをオフにすることを明示できます（-K-とすると，unsigned charオプションはオフであることが明示されます)。Turbo Cのコマンドラインオプションについては付録Bで詳しく解説しています。

コマンドライン上のファイル名

コマンドラインにオプションを並べたあとには，ファイル名を指定します。TCCは，以下の規則にしたがってファイルを処理します。

filename	filename.c をコンパイルする
filename.c	filename.c をコンパイルする
filename.xyz	filename.xyz をコンパイルする
filename.obj	リンク時にオブジェクトとして取り込む
filename.lib	リンク時にライブラリとして取り込む
filename.asm	TASM を呼び出して.OBJ を生成する

このあと TCC はリンカを呼び出して，適切なスタートアップファイルと標準のライブラリの名前をリンカに指示します。

実行可能ファイル

作成される実行可能ファイルの名前は，通常コマンドラインで最初に指定したソースファイルまたはオブジェクトファイルの名前が使用されます。その名前の拡張子を.EXE に置きかえたものが，実行可能ファイルの名前になります。

実行可能ファイルに別の名前を指定したい場合は，-e オプションを使ってください。tcc の後ろ，他のファイル名より前に-e を入力し，続けて（空白を入れずに）実行可能ファイルに指定する名前を入力します。

コマンドラインの例

DOS のコマンドラインから Turbo C を呼び出す適切な入力例を紹介しましょう。

```
tcc -IB:¥include -LB:¥lib -etest start.c body.obj end
```

このコマンドラインの例では，まず DOS のプロンプトから，コマンtccドによって Turbo C が呼び出されます。次に Turbo C は，コマンドラインのオプションを以下のように解釈します。

■インクルードディレクトリは B:¥INCLUDE である（-IB:¥include）
■ライブラリは B:¥LIB ディレクトリの中にある（-LB:¥lib）
■実行可能ファイルは TEST.EXE という名前で作成する（-etest）

さらに Turbo C は，オプションの後に並んだファイル名を以下のように解釈します。

■ START.C というソースファイルをコンパイルする
■ BODY.OBJ というオブジェクトファイルはリンク時にとりこむ
■ END.C というもう1つのソースファイルをコンパイルする

Turbo C のコンパイル時のコマンドラインの例をもう1つ紹介します。

```
tcc -IB:¥include -LB:¥lib2 -mm -C -K s1 s2.c z.asm mylib.lib
```

このコマンドラインは，Turbo C に以下のような指示を与えます。

■インクルードディレクトリは B:¥INCLUDE である（-IB:¥include）
■ライブラリは B:¥LIB2ディレクトリの中にある（-LB:¥lib2）
■メモリモデルはミディアムを使用する（-mm）
■コメントのネストを可能にする（-C）
■ char 型のデフォルトは unsigned（符号なし）とする（-K）

ファイル名のリストは，以下のように解釈されます。

■ S1.C と S2.C というソースファイルをコンパイルする
■ Z.ASM というファイルは TASM を使ってアセンブルする
■実行可能ファイルの名前は S1.EXE とする
■ライブラリファイル MYLIB.LIB をリンク時にとりこむ

TURBOC.CFG ファイル

TCC に指示するオプションの並びは，TURBOC.CFG というコンフィギュレーションファイルの中に書いておくことができ，これはコマンドラインで指定したオプションに加えて TCC が使用します。このコンフィギュレーションファイルには，コマンドラインで入力するのと同じオプションを入れておくことができます。

よく使うオプションをTURBOC.CFGの中に書いておけば，TCC.EXEを使うたびにそれらをタイプする必要がなくなります。TURBOC.CFGに書かれているオプションのどれかを使いたくないときには，コマンドラインからそれを無効にすることができます。

TURBOC.CFGファイルは，一般のテキストエディタやワードプロセッサ（統合開発環境のTurboエディタなど）を使って作成します。各オプションは，空白で区切って同じ行の中に並べても，1行ずつにわけてもかまいません。このファイルを作っておくと，コマンドラインからプログラムをコンパイルする際には，Turbo Cはコマンドラインで指定したオプションに加えて，TURBOC.CFGの中のオプションを使用します。

TCCを起動すると，まずカレントディレクトリにTURBOC.CFGがあるかどうか探します。見つからなければ，（MS-DOS 3.xのもとであれば）次にスタートディレクトリ（TCC.EXEがあるディレクトリ）を探します。TURBOC.CFGは，統合環境版のコンフィギュレーションファイル（TCCONFIG.TC）とは別のものだということを憶えておいてください。

コマンドライン上と，TURBOC.CFG内の同じオプションについては，コマンドラインで指定したものが優先されます。コマンドラインのオプションが優先されるというのは，たいへん重要なことです。たとえば，コンフィギュレーションファイルのオプションの中に-aオプションがあるとします。コンフィギュレーションファイル内の他のオプションは生かして，-aオプションはオフにしたいという場合には，コマンドラインで-a-を指定すればよいわけです。

コマンドラインオプションとTURBOC.CFGのオプションは，どのように組み合わされ，またどのように優先順位が決められるのでしょうか。TURBOC.CFG内のオプションは，その扱われ方によって2つにわけることができます。

■-Iオプションと-Lオプション
■それ以外のすべてのオプション

どんな状況でも，コマンドラインオプションは左から右へ順に評価されます。このとき，次の規則が適用されます。

■-Iおよび-Lを除くすべてのオプションは，複数指定されている場合には右側にあるものが有効になります（したがって右側にあるオフスイッチは，左側にあるオンスイッチを無効にします）。
■左側にある-Iおよび-Lオプションは，右側にあるものより優先されます。

コンフィギュレーションファイル内のオプションがコマンドラインオプションと組み合

わされるときには，TURBOC.CFG 内の-I と-L オプションはコマンドラインオプションの右側に付け加えられ，TURBOC.CFG の残りのオプションはコマンドラインオプションのリストの左側（tcc コマンドの直後）に挿入されます。

コマンドラインと TURBOC.CFG のオプションがこのように組み合わされるので，TURBOC.CFG の-I と-L オプションは一番右端に置かれ，したがって Turbo C は，コマンドラインで指定されたインクルードおよびライブラリディレクトリを先に探すことになります。これによって，コマンドライン上で指定した-I と-L ディレクトリがコンフィギュレーションファイルより優先されます。TURBOC.CFG の他のすべてのオプションはコマンドラインオプションの左側に挿入され，やはりコマンドラインオプションが優先されることになります。

TCCONFIG.EXE：
コンフィギュレーションファイル変換ユーティリティ

統合環境版とコマンドライン版のコンパイラは，付録 B の**表 B.2**にもあるように，多くの共通なオプションを持っています。TCCONFIG.EXE は，一方のコンパイラで作られたコンフィギュレーションファイルをもう一方で使えるように変換します。

変換コマンドの形式は次のようになります。

```
TCCONFIG  SrcFile  [DestFile]
```

TCCONFIG は自動的に変換の方向を判断します。つまり，変換する元のファイル（*SrcFile*）が統合環境（TC）のコンフィギュレーションファイルか，コマンドライン版（TCC）のものかを調べるわけです。

変換先のファイル名（*DestFile*）は省略することができます。ファイル名を指定しなかった場合には，変換の方向に応じて TCCONFIG.TC あるいは TURBOC.CFG というデフォルトのファイル名が使用されます。別のファイル名を指定することもできますが，コマンドラインコンパイラ（TCC）がコンパイル時に調べるのは TURBOC.CFG というファイル名だけです。他のファイルを使用することはありません。

TCCONFIG が TCCONFIG.TC を作成する際，コマンドライン版（TCC）のコンフィギュレーションファイル（TURBOC.CFG）で指定されていない項目については，TCCONFIG はデフォルトの値を使います。逆方向の変換の際には，TCCONFIG.TC のオプションの中でデフォルト値と違うものだけが TURBOC.CFG に取り込まれます。

TCCONFIG による変換が終了すると，MS-DOS のプロンプトに戻ります。

MAKEユーティリティ

Turbo　CのスタンドアロンのMAKEユーティリティは，統合環境のプロジェクトMakeの機能強化版で，ソースファイルとオブジェクトファイルの依存関係を記述することができます。これはUNIXのmakeユーティリティに基づいています。MAKEユーティリティは依存関係を評価して，ファイルが正しくコンパイルされリンクされることを保証します。

MAKEユーティリティを用いる利点はどんなところにあるのでしょうか。プロジェクトMakeと同じように，前回コンパイルしたときから，プログラムを構成するモジュールのどれが変更されたかを意識する必要がありません。それだけではなく，スタンドアロンのMAKEユーティリティは汎用のプログラム作成ツールであり，プロジェクトMakeよりさらに強力な機能を持っています。MAKEは，こみいった構成のプログラムをリンクする前に，更新すべきファイルをすべて再コンパイルします。そのあと，新しくコンパイルしたオブジェクトファイルを，再コンパイルする必要のなかったオブジェクトとリンクして，新しい実行可能ファイルを作り出します。

スタンドアロンのMAKEユーティリティについては，付録Cで詳しく解説しているので，詳細についてはそちらを参照してください。

BUILTINS.MAK

BUILTINS.MAKは，何度も繰り返し使うようなMAKEのマクロとルールを収めておくオプションファイルです。これを作っておけば，同じマクロやルールをそのたびにタイプする必要がなくなります。makefileの書き方やBUILTINS.MAKファイルのセットアップについては付録CのMAKEユーティリティの節を参照してください。

コマンドラインからTurbo Cのプログラムを走らせる

MS-DOSのコマンドラインからTurbo Cの実行プログラムを走らせるには，プロンプトから単にそのファイルの名前をタイプするだけです。拡張子.EXEはつけません。たとえば，プログラムTEST.EXEを実行するには，MS-DOSのプロンプトで，testとだけタイプして**リターン**を押します。これでTESTプログラムが実行されます。

先へ進もう

ここまで，統合開発環境とコマンドラインの両方について，Turbo C のプログラムをどうやってコンパイル・リンクして走らせるかを見てきました。もうすでに，Turbo C の能力を試してみる準備は整っています。これから，言語およびこの処理系の知識が広がっていくにつれて，2冊目のマニュアル**『Turbo C リファレンスガイド』**を参照する機会がふえてくるでしょう。リファレンスガイドでは，実行時の環境，ライブラリファイル，さらにC 言語が Turbo C でどのように実現されているか，などについてくわしく解説されています。それでは先へ進んでいきましょう。第4章では Turbo C 統合デバッガを使う練習，第6章と第7章ではプログラミング言語 C の概要，第8章では Turbo C のグラフィックス関数の紹介，第12章では高度なプログラミングテクニックが述べられており，あなたの到着を心待ちにしています。

Turbo Pascal や Turbo Prolog を使ったことのある方は，それぞれ第9章あるいは第11章を読んでみてください。これらの言語とこの高速かつ強力な Turbo C を，どうすればうまく組み合せて使えるかについてヒントが得られるでしょう。

第4章

プログラムのデバッグ

プログラムのコンパイルエラーやリンクエラーを訂正して実行してみても，初めは正しく動かないことがよくあります。新しいプログラムには，必らずたくさんの**バグ**――プログラムの設計上・コーディング上のエラー――があり，それを発見して修正しなければなりません。プログラムのバグの発見と修正の作業を**デバッグ**と呼びます。

プログラムの動作を見るだけではバグの発見は困難なことも多く，このためほとんどのプログラマは，プログラム中のバグをつきとめるためにデバッガを利用します。デバッガとは，プログラムの動作を制御しながら実行させることができるソフトウェアです。プログラムの実行を任意の場所で止めたり，1ステートメントずつ実行したり，処理中のデータの内容を調べたりすることができます。

この章では……

Turbo Cの統合環境は，統合デバッガと呼ばれるデバッカを内蔵しています。この章ではTurbo C統合デバッガをどのように使うのかを説明します。

この章は，統合デバッガの使用法を説明するための一連の例から始まります。最初の例では，"簡単な"バグを発見するための，デバッガの最もやさしい機能の利用法を示します。その後の例では，デバッガのより高度な機能を示します。

次に，デバッガのメニューコマンドと，それに関連するメニューコマンドのあらましを，対応するホットキー，あるいはホットキーの組み合わせとともに述べ，各コマンドが行なうことの記述が続きます。

最後に，デバッグをより容易に行なうための指針をいくつか示します。指針の多くは，デバッグの方法だけでなく，プログラムの設計法・記述法にも関連しています。こうした部分の多くは，Turbo Cだけでなく，他のプログラム言語に対しても適用することができるはずです。

統合デバッガの働らき

Turbo C の統合デバッガは**ソースレベルのデバッガ**です。このことは，プログラムを記述したのと同じ“言語”を使って，デバッガを制御することができることを意味しています。たとえば，配列中の要素の値を表示するには，次のような式で値を表示するようにデバッガに指示することができます。

```
rptr -> image[nptr+0x80]
```

プログラムのデバッグは単に，**Run/Run** メニューオプション（ホットキー ***CTRL-F9***）を用いてプログラムを走らせるだけで行なうことができます。**Source debugging** トグルが **On** の状態でプログラムをコンパイルしてあれば，引き続きデバッガが起動されます（このトグルを設定するには **Debug/Souce debugging** を選択します）。

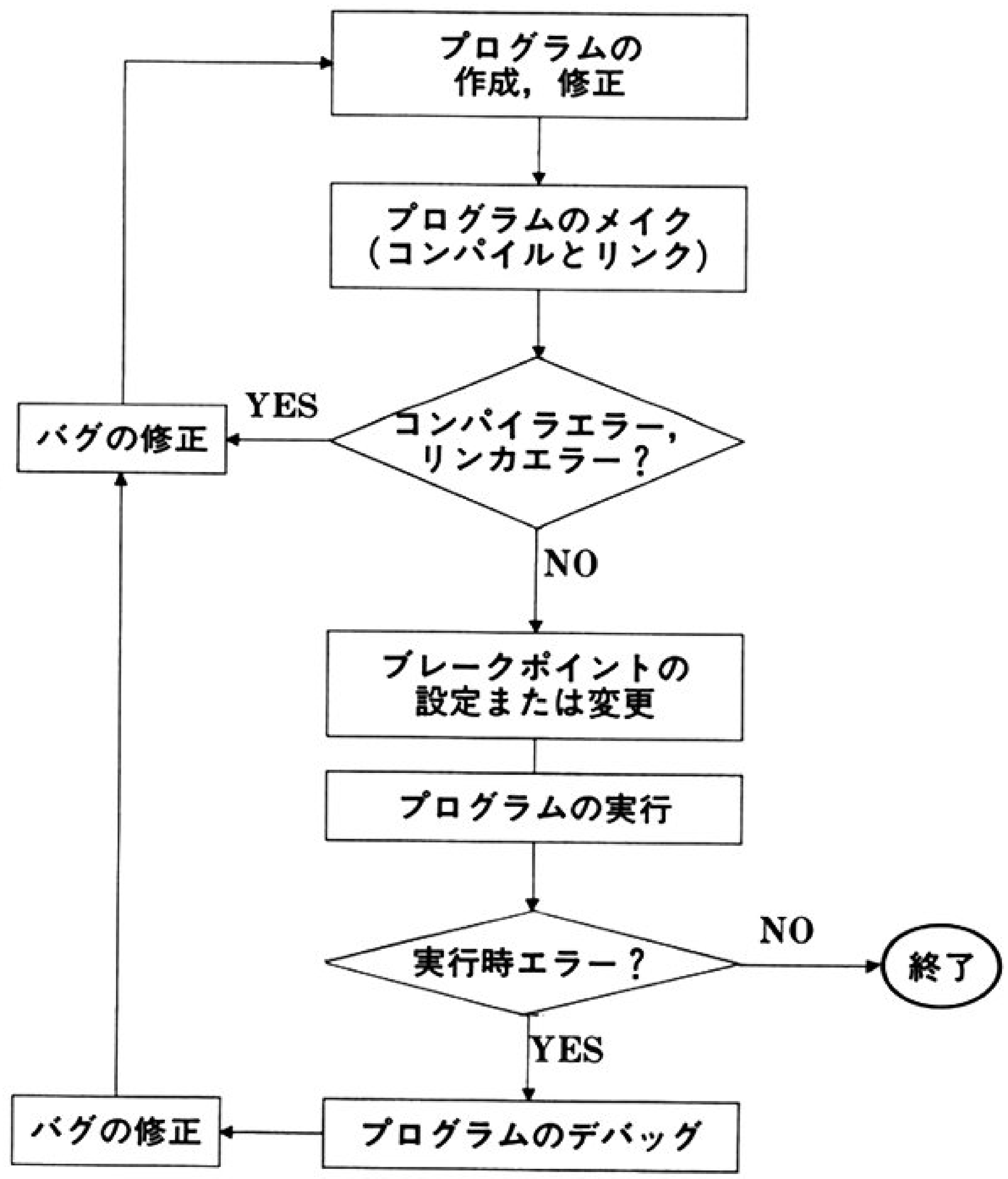
プログラムを実行する前に，ソースファイル中のいくつかの行にブレークポイントを設定することができます。実行中のプログラムの制御がブレークポイントに到達すると，ブレークポイントのある行の最初の文を実行する直前でデバッカは一時停止し，制御をユーザに戻します。

プログラムの停止中には，プログラムをいろいろと調べることができます。たとえば次のようなことが可能です。

■変数や式の値を表示する。
■特別なウインドウに式のリストをセットし，それらの値の変化のしかたを見る。
■変数の値を変更する。
■ブレークポイントの除去，あるいは新しいブレークポイントの設定。
■プログラムを1行ごとに実行する。
■ファイルの編集，プログラムの再コンパイルと再リンク，Turbo C のメニューシステムの他の機能の利用。
■別のブレークポイントに到達するまでプログラムを実行する。

図 4.1 にデバッグの際の典型的な作業を示しています（この図の各ステップが統合デバッガによって行なえるすべてのことを示しているわけではないので注意してください）。

図 4.1　デバッグ過程における典型的なステップ



演習1：簡単なプログラムのデバッグ

Turbo C での最初のデバッグ体験として，次に示したプログラムを使用することにします。このプログラムを WORDCNT と呼ぶことにしましょう。これはテキストファイルの内容を表示し，単語の長さの表を作ります。つまり，1文字からなる単語の数，2文字からなる単語の数，・・・・・を示すのです。ところがあいにく，この WORDCNT にはバグがいくらか含まれています。デバッガを用いて，バグをみつけることにしましょう。

マスターディスクの中のファイル WORDCNT.C の中にプログラム WORDCNT が入っています。これを自分の Turbo C ディレクトリにコピーして，バグ入りのプログラムファイルが確かにあるようにしてください。

Turbo　C が含まれているディレクトリ以外で作業をする場合は，そのディレクトリに WORDCNT.C の作業用コピーを作ってください。また，プロジェクトファイル WORDCNT.PRJ とデータファイル WORDCNT.DAT の作業用コピーも作ってください。これらのファイルは3つとも，マスターディスクおよび Turbo C ディレクトリにあります。

```
/*****
*  WORDCNT.C - Sample program for the debugging tutorial
*
*  Copyright (c) 1988 Borland International. All rights reserved.
*
*  NOTE: This program is for use with the debugging tutorial
*        in the debugging chapter of the User's Guide. It
*        intentionally contains bugs.
******/

#include <stdio.h>
#include <ctype.h>

#define MAXWORDLEN      16
#define NUL             ((char)0)
#define SPACE           ((char)0x20)

/*****
* Find the next word in the line buffer.
* IN:     wordptr points to the first character of a word or a preceding
*         space.
* RETURN: A pointer to the first character of the word. If there are no
*         more words, a pointer to the terminating NUL.
*****/
char *nextword(char *wordptr)
{
/* Advance to the first non-space. */
  while ( *wordptr==SPACE )
    wordptr++;
  return(wordptr);
}
```

```
/*****
* Find the length of a word. A word is defined as a sequence of characters
* terminated by a space or a NUL.
* IN:    wordptr points to a word.
* RETURN:The length of the word.
*****/
int wordlen(char *wordptr)
{
  char *wordlimit;
  wordlimit = wordptr;
  while ( *wordlimit & *wordlimit!=SPACE )
    wordlimit++;
  return( wordlimit-wordptr );
}


/*****
* The main function.
*****/
void main(void)
{
  FILE  *infile;          /* Input file. */
  char  linebfr[1024],    /* Input line buffer, very long for safety. */
        *wordptr;         /* Pointer to next word in linebfr. */
  int   i;                /* Scratch variable. */
  static int  wordlencnt[MAXWORDLEN],
                          /* Word lengths are counted in elements 1 to
                           MAXWORDLEN. Element 0 isn't used. The array is
                           static so that the elements need not be set to
                           zero at run time. */
              overlencnt;       /* Overlength words are counted here. */

  printf("WARNING: This is an example program for the practice¥n");
  printf("debugging session.  If you are not running this under the¥n");
  printf("Integrated Development Environment press control-break now.¥n");
  printf("See the debugging chapter of the User's Guide for details.¥n¥n");

  printf( "Enter the input file's name: " );
  gets(linebfr);
  if ( !strlen(linebfr) ) {
    printf( "You must specify an input file name!¥n" );
    exit();
  }

  infile = fopen( linebfr, "r" );
  if ( !infile ) {
    printf( "Can't open input file!¥n" );
    exit();
  }

/* Each loop processes one line. NOTE: if a line is longer than the
 input buffer the program may produce invalid results. The very large
 buffer makes this unlikely. */
```

```
  while (  fgets( linebfr, sizeof(linebfr), infile )   )   {
    printf("%s¥n",linebfr);
    /* Check for buffer overflow & remove the trailing newline. */
    i = strlen(linebfr);
    if ( linebfr[i-1] != '¥n' )
      printf( "Overlength line beginning¥n¥t%70s¥n", linebfr );
    else
      linebfr[i-1] = NUL;

    /* lineptr points to the 1st word in linebfr (past leading spaces). */
    wordptr = nextword(linebfr);

    /* Each loop processes one word. The loop ends when [nextword]
       returns NULL, indicating there are no more words. */
    while (*wordptr) {
    /* Find the length of this word, increment the proper element of the
       length count array, & point to the space following the word. */
      i = wordlen(wordptr);
      if ( i > MAXWORDLEN )
        overlencnt++;
      else
      ;
      wordlencnt[i]++;
    wordptr += i;

      /* Find the next word (if any). */
      wordptr = nextword(wordptr);
    }
  }

/* Print the word length counts. Each loop prints one. */
  printf(  "  Length Count¥n" );
  for ( i=1; i<MAXWORDLEN; i++ )
    printf( "  %5d %5d¥n", i, wordlencnt[i] );
  printf( "Greater %5d¥n", overlencnt );

/* Close the file & quit. */
  fclose(infile);
}
```

Debug/Souce debugging と **O/C/C/OBJ** debug information トグルが両方とも On になっていることも確認してください。DOS のプロンプトに対して次のように入力して，TC を起動してください。

```
TC WORDCNT
```

そしてプログラムを作成してください（Compile/Build all を選択します。Turbo C は WORDCNT をコンパイル，リンクして実行可能ファイルになるようにして停止します。そこで **R**un/**T**race into を選択します（あるいは ***F7*** を押します）。

この時点で，デバッガは **main** の先頭までスクロールして，Edit ウィンドウに入ります。関数 **main** の宣言（void main(void)）を含む行がハイライト表示されていますが，これ

はWORDCNTを走らせた場合に最初に実行される行であることを表わしています。このハイライト表示を**実行バー**と呼び，**実行位置**，すなわち次に実行される文を含む行を示すマークになっています。

WORDCNTを走らせるには，**Run**/**Run**を選択します。するとWORDCNTは入力ファイルの名前を聞いてきます。WORDCNT.DATとタイプして**リターン**を押してください。WORDCNTはデータファイルの最初の行を読み込んで表示しますが，プログラムのバグのためにそこで止まってしまいます。***CTRL-STOP***を押してそれを解除し，***ESC***を押して確認してください。そこでRun/Program resetを選択して（あるいはホットキー***CTRL-F2***を押して）デバッグを終え，***F7***（Run/Trace into）を押して，新たにデバッグを開始してください。

ブレークポイントの設定と利用

ソースファイルのコメントでの説明にもある通り，**main**の最初の部分は，入力ファイルの名前を要求し，それをオープンします。WORDCNTが，入力ファイルの第1行を読み込んで表示したことは，コードのこの部分は正しく動いていることを強く示しています。したがって，**main**の最初の部分は通過して，あやしい部分に近づいた時点で実行を止めることにします。そのためには，実行を止めたい行の上に**ブレークポイント**を設定しなければなりません。

ROLLUP（***PgDn***）と下矢印キーを使って，

```
while ( fgets( ...
```

で始まる行（Each loop processes ...で始まる長いコメントの直後）に，カーソルを移動します。実行バーは動かないことに注意してください。プログラム中の文を実行させたのでなく，単にエディタのカーソルを動かしただけだからです。

カーソルのある行の上にブレークポイントを設定するには，**B**reak/watch/**T**oggle breakpoint（あるいはホットキー***CTRL-F8***）を選択します。すると，デバッガはその行をハイライト表示し，そこにブレークポイントが設定されたことを示します。このブレークポイントのハイライト表示のようすは，実行バーのそれとは異なる点に注意してください。

ここで，**Run**/**Run**選択してください（あるいはホットキー***CTRL-F9***を押します）。WORDCNTは実行を続行（この場合は開始）します。

WORDCNTは入力ファイルの名前を要求した後，停止してキーボード入力を待ちます。デバッガは実行画面を表示することによって，WORDCNTが必要とするものを示します。実行画面には，あたかもデバッガなしでプログラムが動いているかのように，プログラムの出力が表示されます。次にWORDCNT.DATとタイプしてリターンを押してください。

WORDCNT はブレークポイントに達するまで走り続けます。ブレークポイントまで来るとそこで実行を停止し, Edit ウィンドウが再び現われます。カーソルと実行バーは while 文の上にきています。

while 文の上に設定したブレークポイントは, 相変わらず存在していることに注意してください。実行バーがあるのでブレークポイントはかくされて見えませんが, 実行バーが移動すれば再び見えるようになります。このブレークポイントをオフにしない限り, ここに到達するごとに WORDCNT は停止します。

CTRL-STOP の利用

設定が可能なブレークポイントの他に, 実行中に作ることができる"即席"ブレークポイント (***CTRL-STOP***, IBM PC では ***Ctrl-Break***) もあります。すなわち, クラッシュを除けば, 任意の時点でプログラムに割り込みをかけることができるのです。***CTRL-STOP*** を押せば, 実行中のプログラムから抜けて TC エディタに戻ります。このとき, 次に実行される行の上に実行バーが表示され, プログラムの残りの部分が実行できるようになっています。

実際には, 以下のことが行なわれています。デバッガは DOS, BIOS, 他のサービスと関連しているので, 現在は DOS ルーチンの実行中か, BIOS ルーチンの実行中か, プログラム自身の実行中かを知っています。***CTRL-STOP*** が押されると, デバッガは, プログラム自身のコードが実行中になるまで待ちます。それから, C のソースコード行の先頭に対応する機械語命令に到達するまで, 機械語命令をひとつずつ実行します。そこでブレークが行なわれて, 実行バーがコードの該当する行にくるわけです。

デバッガがソースコードの行の位置を決定して表示する前に, さらにもうひとつの ***CTRL-STOP*** を検出した場合は, プログラムの実行を即座に終了し, ソースコードの行の位置の決定は行ないません。このときは残っている出力のはき出しや終了関数の呼び出しを行なわずに, プログラムは終了します(関数 **_exit** によって実行を終了する場合と似ています)。したがって, ***CTRL-STOP*** を2回押すことは, プログラムが無限ループに入ってしまったときなど, 1回 ***CTRL-STOP*** を押しただけでは中断することができない場合にのみ行なうべきです。

関数呼び出しのステップ実行

さて, WORDCNT のバグのありそうな部分にさしかかったので, さらに注意深く進まなければなりません。**main** の残りの部分が意図したように動いているかどうかをみるために, 各行のおわりで停止しながら1行ずつ実行していきましょう。

main の1行を実行するには, **Run/Step over** を選択します。デバッガは, while 文の実

行にかかり，入力の最初の行を読み込みます。それから，ハイライト表示を次の実行可能な文を含む行に移します。続く **printf** の呼び出しを実行するには，再び Run/Step over を選びます。

while 文は，意図通りの動きをしているでしょうか？これを調べるには，Run メニューで User screen を選択するか，あるいは ***GRPH-F5*** を押します。このコマンドは，実行画面を表示します。この画面には入力の最初の行が見えているので，while と **printf** は正しく動いていると結論してもよさそうです。任意のキーを押すと，Edit ウィンドウが再び表示されます。

デバッガには，Run/Step over に対するホットキー（***F8***）があります。***F8*** を押すと，次の文が実行されます。この文では，入力行の長さが計算され，それが変数 *i* に代入されます。

警告!ライブラリ関数 **longjmp** の呼び出しを Trace into または Step over しても次の行では止まりません（この関数から戻ってくることが決してないからです）。すると，次のブレークポイントまで，あるいは実行が終了するまで，プログラムは走ることになります。

式の評価

代入文は，意図した通りの結果をもたらしたでしょうか？ *i* の値が表示されれば，これを確かめることができます。

Debug/Evaluate を選択してください。デバッガは，次に示すような3つのフィールドを持つポップアップウィンドウを開きます。

1. **Evaluate（評価）フィールド**：ここには，評価したい，つまり場合によっては変更したい式を入力します（**注意**：入力したい式が長くて Evaluate フィールドに入りきらないときは，左右の矢印キー，***HOME***，***SHIFT-HOME***（***End***）を用いて式をスクロールさせることができます）。
2. **Result（結果）フィールド**：デバッガは，式の値をここに表示します。
3. **New value（新しい値）フィールド**：式に保持させたい新しい値をここに入力します（オプション）。

Evaluate フィールドには，語が1つ含まれている点に注意してください。この語は Evaluate フィールドのデフォルト値で，Edit ウィンドウ中のカーソル位置にある語をコピーしたものです。デフォルト値の利用については少し後で考えることにします。当面は，式 *i* をそこに入力して，**リターン**を押してください。デバッガは Result フィールドに *i* の値を表

示します。43と正しい値になっています。***F10*** を押して，メニューシステムから抜け出してください。

次の文のグループでは，**fgets** で読み込んだ行が改行文字で終わっているかを確認します。改行文字で終わっていなければ，入力行がバッファに収まる長さよりも長かったことになるので，プログラムは警告メッセージを表示します。行が改行文字で終わっていれば万事うまくいっているわけで，改行文字が単語の最後の文字として数えられないように取り除きます。

if 文を実行する前に，いま入力した行を表示して，どのような結果が期待できるかをみておくのもよいことです。それには，Edit ウィンドウ中の語 *linebfr* のところにカーソルを動かしておいて Debug/Evaluate をもう一度選択してください（またはホットキー ***CTRL-F4*** を押す）。デバッガは，Debug/Evaluate ウィンドウの Evaluate フィールドに *linebfr* と表示します。そこで**リターン**を押してください。

デバッガは，次のように表示するはずです。

```
To be or not to be: that is the question.¥r
```

¥n は，C のソースプログラムと同じく，改行文字を表わしています。

ここで ***F8*** を押して if 文を実行してください。実行バーは else 節に移動するので，if 文は正しく動いたことになります。else 節中の代入文を実行し，再び linebfr を評価してください（Debug/Evaluate のかわりにホットキー ***CTRL-F4*** を使うことができます）。改行文字は除かれているので，ここまでは正しいことがわかります。

関数 nextword と wordlen

次の文では *nextword* を呼んでいますが，この関数は文字列の中の次の（この場合は最初の）語を示すポインタを返します。***F8*** を押してこの文を実行して *wordptr* を評価し，*nextword* が返した値を見てください。*wordptr* は To be or not to be...の *T* を指しているはずです。したがって，少なくともこの簡単な場合には，*nextword* は正しく動いていることになります。

次にプログラムは while ループに入ります。ループの各繰り返しでは，*linebfr* 中の語1つを処理し，*wordptr* を次の語に進めるはずです。そして，ループで最後の語を処理した後は，*wordptr* は行の終わりにあるヌル文字を指して，ループが終了するはずです。

while 文を実行してください。実行バーはループ本体中の最初の文（関数 *wordlen* を呼んでいる）に移ります。この関数は，*wordptr* が指している語の長さを決定します。この文を実行し，*i* を評価してください。値は0と表示されますが，これは正しくありません。行の最初の語の長さは2でなければなりません。バグを1つ発見しました!

ひと呼吸して考えよう

バグを取り除く前に，そのバグがプログラムに与える影響を考えてみるのも価値のあることです。語の長さが0であるというエラーは，2つの影響を与えます。第1に，誤った *word-lencnt* の配列要素（0番目）をインクリメントしてしまうことになります。

第2に，wordptr ＋＝ i ；を実行しても *wordptr* の値は変化せず元のままです。したがって，while ループの2回目の繰り返しの始まりでは，*wordptr* の値は1回目と同じ値になります。*wordlen* は1回目の呼び出しで0を返したので，2回目もおそらく0を返すだろうと考えられます。そのため，while ループの3回目でも *wordptr* の値は同じになるでしょう。4回目もその後も，いつまでも同様のことが続きます。これは，WORDCNT の実行のようすと完全に符合します。これが WORDCNT をハングアップさせたバグなのです。

この調査でのポイントはなんだったのでしょうか。WORDCNT のおかしな動作のごく一部を示すバグを見つけただけかもしれませんし，あのおかしな動作とはなんの関係もないことがわかったのかもしれません。いずれの場合にせよ，もっとプログラムを動かして，次になにが起こるかを見てみたいところでしょう。それによって，発見したバグを修正することに集中でき，プログラムが正しい（少なくとも改善された）動作を示すという信頼性を高めることができます。

今までに行なったこと

WORDCNT がうまく実行されないのは，*wordlen* 中のバグによるものであることがわかりました。次に，*wordlen* のどこがおかしいかを見つけなければなりません。その点についてはすぐこのあとで取り組むことにして，その前に，ここまでで使ったデバッガコマンドを復習し，さらに各コマンドについて詳しく学んでおきましょう。

WORDCNT のデバッグでは，次のことを行ないました。

- **Debug/Souce debugging** と **O/C/C/OBJ debug information** が On になっていることを確認しました。
- WORDCNT のデバッグの準備のために **Compile/Build all** を選択しました。
- エディタコマンドを用いてカーソルを WORDCNT のあやしいと思われる部分に移動し，**Break/watch/Toggle breakpoint** を選択して，そこにブレークポイントを設定しました。**Run/Run**（またはホットキー ***F9***）を選択して，WORDCNT をブレークポイントまで実行しました。
- **Run/User Screen**（またはホットキー ***GRPH-F5***）を使い，プログラム出力をユーザ

画面に出力しました。

■ Run/Step over（またはホットキー ***F8***）を選択し，**main** 中の文を1行ずつ実行しました。

■ Debug/Evaluate（またはホットキー ***CTRL-F4***）を選択し，いくつかの変数の値を表示させました。

■発見したバグについて考え，それによって WORDCNT のおかしなふるまいの説明がつくと結論づけられたので，この後ただちにバグを修正することにしました。

Evaluate ウィンドウ内のデフォルト式

Debug/Evaluate を選ぶと，Edit ウィンドウのカーソル位置の語が Evaluate フィールドの中にコピーされたことを思い出してください。Debug/Evaluate を選ぶ前に，評価したい変数にカーソルを置いておくことにより，しばしば手間を省くことができます。たとえ必要となる式とは違っていても，デフォルト式を編集すれば，式全体をそっくりタイプするよりもすばやくその式を入力できるはずです。さらに，右矢印キーを使えば，Edit ウィンドウからデフォルト式に，もっと多くのテキストをコピーすることができます。右矢印を押すたびに，次の文字が1文字ずつコピーされていきます。

たとえば，ソースファイルの次の行の中の式 *linebfr*[*i*-1]を評価したいとします。

```
if ( linebfr[i-1] != '¥n' )
```

カーソルを *linebfr* に移動して Debug/Evaluate を選んでください。Evaluate フィールドには，*linebfr* というデフォルト式が表示されます。その式に[*i*-1]も付け加えるために，右矢印キーを5回押してください。そして**リターン**を押します。

評価した式の値の変更

Debug/Evaluate では，いくつかの型の式においては，その値を変更することができます。*i*，*linebfr*[*i*]，＊(*linebfr*＋*i*)のような1つのデータ要素を表わす式ならすべてその値を変えることができます。

変数 *i* を評価した後，その値を変えてみてください。*i* を評価するために**リターン**キーを押すと，デバッガは Result フィールドに *i* の値を表示します。New value フィールドに（↓キーを使って）移動し，*i* に代入したい値を入力してください。たとえば，New value フィールドを編集して *i*+1としたり（*i* を1だけ増す），17と入力したりします。**リターン**キーを押すと，デバッガは入力されたものを評価して *i* の値を変更し，Result フィールドに新しい値を表示します（**注意**：New value ボックス内で1度**リターン**を押してしまうと，

ESC キーを押しても変更を取り消すことはできないということを憶えておいてください)。***ESC*** キーを押して **Debug/Evaluate** から一度出て再び同じコマンドを入力し，*i* の値を再表示して確かに変更されていることを確かめてください。それから *i* の値を元に戻し，**Debug/Evaluate** から抜けます。

あるバグによって引起こされた影響を修正するために式の値を修正することができ，それによってプログラムを少し先まで走らせることができるので，より多くのバグを見つけることもできます。また，プログラムの実行のようすを見きわめるために利用することもできます。たとえば，ある関数に妥当でない引数が渡されたときに，その関数がどのように動作するかを知りたいとしましょう。プログラムが特定の値を関数に渡すようにするのは難しいこともありますが，プログラムが関数を呼び出す直前に，変数の値を変えてやれば同じ結果を得ることができます。

リターンキーのかわりに ***ESC*** キーを押して New value フィールドから出ると，デバッガは式の値を変更しません。間違って New value フィールドに入力してしまった場合や，値を入力してから気が変わった場合は ***ESC*** キーを押してください。

次に示すものを含んでいなければ，通常の C 言語の式ならどんな式でも評価できます。

- ■関数の呼び出し
- ■# define または typedef で定義されたシンボルやマクロ(＊ *wordptr* == 0x20は許されますが，＊ *wordptr* == SPACE は，SPACE が# define で定義されているので許されません)
- ■実行されている関数のスコープ内で"見えない"ローカル変数または静的変数で完全修飾されていない場合

変数名の修飾

式の中で用いられる変数名を完全に修飾することが望ましい典型的な場合が2つあります。

- ■いろいろなモジュールの静的変数を調べたいとき
- ■関数の自動（ローカル）変数または静的変数を見たいとき

変数名を完全に修飾するには，次の構文を使用します。

.<モジュール名>.<関数名>.<変数名>

モジュール名や関数名は，場合によっては省略されてもよい点に注意してください。た

とえば，**main** 関数をステップ実行していて，*mysubs* という別のモジュール内にある *myvar* という静的変数を見たい場合，. *mysubs.myvar* とタイプすることができます。しかし，*myvar* がモジュール *mysubs* 内の関数 *myfunc* 内にある場合は，. *mysubs.myfunc.myvar* と指定しなければなりません。一方，*myfunc* が関数 **main** と同じモジュール内にあるなら，. *myfunc.myvar* とだけタイプすればよいのです。

フォーマット指定子

情報を Debug/Evaluate ウィンドウの中でどのように表示するかということを正確に制御するために，Turbo C では，Evaluate フィールド内の式に（オプションの）**フォーマット指定子**を付け加えることができます（Watch ウィンドウにも同様のことが当てはまります）。フォーマット指定子は，式の終わりにカンマで区切って付け加えます。大文字小文字どちらで書いてもかまいません。

フォーマット指定子は，オプションである**繰り返し回数**（整数）の後ろにフォーマット文字をいくつかつけたものから成り立っています。繰り返し回数とフォーマット文字の間に空白を入れてはいけません。**表 4.1** に使用可能なフォーマット文字とその効果が示されています。

繰り返し回数は連続した変数を表示するのに使用されます。その典型例は配列の要素の場合です。たとえば，*list* を10個の整数配列とすると，式 *list* を指定すると，次のように表示されます。

```
{ 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 }
```

配列の特定の範囲を見たい場合は，調べたい最初の要素の添字を指定し，繰り返し回数を付け加えます。

```
list[5],3
```

こうすると Result フィールドには次のように表示されます。

```
60, 70, 80
```

このテクニックは，大きすぎて1行にすべてを表示できない配列を扱う際に有効です。

繰り返し回数は配列に限られているわけではありません。どんな変数の後ろにも繰り返し回数をつけることができます。*var*,⟨*n*⟩という一般的な構文は，単に *var* のアドレスから始まる *var* と同じ型の *n* 個の連続した変数を表わします。ただし，式が変数と同等のものでない場合は，繰り返し回数は無視されることに注意してください。大ざっぱにいえば，代入文の左辺に書くことができるか，あるいは関数への引数として使用することができれ

ば，それは変数であるといえます。

表 4.1　デバッガのフォーマット指定子

文字	機能
C	**文字**。制御文字（ASCII コードの0から31）に対して特別な表示文字を出します。たとえば，^Aは，"ニコニコ顔"として表示されます。これは文字や文字列に対して働きます。
S	**文字列**。制御文字（ASCII コードの0から31）をC言語のエスケープ文字列を使用してASCII コードで表示します。これは，文字や文字列の表示に対するデフォルトのフォーマット指定子であり，M指定子と組み合わされて使用される場合にのみ役に立ちます。
D	**10進**。すべての整数値は，10進で表示されます。単純な整数型の式の他に，整数を含む配列や構造体に対しても働きます。
HまたはX	**16進**。すべての整数値は，0xを前につけて16進で表示されます。単純な整数型の式の他に，整数を含む配列や構造体に対しても働きます。
F⟨*n*⟩	**浮動小数点**。*n* は2から18の間の整数で，表示する有効数字の桁数を示します。浮動小数点値にのみ働きます。
M	**メモリダンプ**。指定された式のアドレスからのメモリダンプを表示します。式は代入文の左辺における構文，つまりメモリアドレスを示すものでなければなりません。そうでない場合M指定子は無視されます。デフォルトでは，変数の各バイトは2桁の16進数で表示されます。M指定子にD指定子をつけると各バイトは10進で表わされ，H指定子またはX指定子をつけると各バイトは16進で表わされます。C指定子またはS指定子が指定されると，変数は（特殊文字を含むものも含まないものも）文字列として表示されます。表示されるデフォルトのバイト数は変数のサイズに対応しますが，繰り返し回数を使って指定してもかまいません。

P

ポインタ。ハードウェア志向のデフォルトの *seg*：*ofs* 型のフォーマットだけではなく，指されているアドレスについての情報を合わせ持った *seg*：*ofs* 型のフォーマットでポインタを表示します。特に，セグメントが置かれているメモリ領域とオフセットアドレスに存在する変数名の名前を示します。メモリ領域を次に示します。

メモリ領域	**Evaluate メッセージ**
0000：0000 － 0000：03FF	Interrupt vector table (割り込みベクターテーブル)
0000：0400 － 0000：04FF	BIOS Data area (BIOS データ領域)
0000：0500 － Turbo C	MSDOS / TSR's (MS-DOS とメモリ常駐プログラム)
Turbo C - ユーザプログラム PSP	Turbo C
ユーザプログラム PSP	User Process PSP (ユーザプロセスの PSP)
ユーザプログラム-RAM の最後部	変数に割り当られたメモリ内にそのアドレスがあれば静的ユーザ変数の名前，そうでなければ何もなし

■ PC-9801 の場合：

A000：0000 － A000：3FFF	Text Video RAM(テキスト VRAM)
A000：4000 － A000：7FFF	Not Used (未使用)
A800：0000 － B000：FFFF	Graphic Video RAM (グラフィック VRAM)
C000：0000 － C000：7FFF	User Enhanced ROM (ユーザ拡張 ROM)
C800：0000 － E000：FFFF	System enhanced ROM (システム拡張 ROM)
E000：0000 － E000：7FFF	Graphic Video RAM (グラフィック VRAM)
E800：0000 － F000：FFFF	N88-BASIC RAM

■ IBM PC の場合：

A000：0000 － A000：FFFF	EGA Video RAM(EGA ビデオ RAM)

	B000：0000 - B000：7FFF	Monochrome Display RAM （モノクロディスプレイ RAM）
	B800：0000 - B800：7FFF	Color Display RAM （カラーディスプレイ RAM）
	C000：0000 - E000：FFFF	EMS Pages/Adapter BIOS ROM's （EMS ページ/アダプタ BIOS ROM）
	F000：0000 - F000：FFFF	BIOS ROM
R	**構造体/共用体**。{ X：1, Y：10, Z：5 }のように値とフィールド名を表示します。構造体と共用体に対してのみ働きます。	

次にフォーマット指定子に関する一般的な規則をいくつかあげます。

1. フォーマット指定子は，それが有効な型の変数に添えられた場合にのみ効果があります。それ以外の場合は無視されます。
   評価される式が複数のオブジェクトを含む場合（たとえば構造体）は，複数個のフォーマット指定子を式の終わりにつけますが，各オブジェクトに対して適切なフォーマット指定子が当てられます。たとえば，整数と実数を含む構造体を表示するためにstruct，F5H と入力したとすると，整数は16進数（H）で，実数は有効数字5桁の浮動小数点フォーマット（F5）で表示されます。
2. ある1つの式に対して同じ型の2個以上のフォーマット指定子を入力した場合，相容れない指定子のどちらが使用されるかは，型に依存する優先順位によって決まります。つまり，優先順位の最も高いものが選ばれます。もちろん，この問題は構造体または共用体に対してだけ起こります。単純変数や配列変数に対しては，普通に1個のフォーマット指定子のみを入力してください。

表 4.2　フォーマット指定子のクラスにおける優先順位とデフォルト

型	指定子の優先順	デフォルト
char	C S H D	S
unsigned char	C S H D	S
int	H D C＊ S＊	D
unsigned int	H D C＊ S＊	D
long	H D C＊ S＊	D
unsigned long	H D C＊ S＊	D
char ポインタ	C S P H	S
他のポインタ	P H	P
enum	H D C＊ S＊	D（メンバ名が後に続く）
float	Fn	F7
double	Fn	F15
long double	Fn	F18
char の配列	C S H D	S
他の配列	{ }により囲まれていてカンマで区切られた要素	
構造体	R	−

＊ 文字型フォーマット指定子は，値が適切な範囲内に入っている場合にのみ受け入れられます（signed 型は-128から127まで，unsigned 型は0から255まで）。

注意：ポインタ変数に対して用いられた H フォーマット指定子は，ポインタを16進整数値として表示します。

フォーマット指定子の使い方を示すために，次のような構造体と変数が宣言されていると仮定しましょう。

```
struct {
    int    account;
    char   name[10];
} client = { 5000, "Jones" };

int    list[5] = { 0, 10, 20, 30, 40 };
int   *ptr = list;

void main()
{
}
```

この場合，Evaluate フィールドに次のような式を入力すると，Result フィールドに対応する評価が示されます。

Evaluate（評価）	Result（結果）
`list`	`{ 0, 10, 20, 30, 40 }`
`list[2],3`	`20, 30, 40`
`list[2],3x`	`0x14, 0x1E, 0x28`
`list,m`	`00 00 0A 00 14 00 1E 00 28 00`
`ptr`	`list`
`ptr,p`	`DS:01A3 list`
`*ptr,3`	`0, 10, 20`
`client`	`{ 5000, "Jones" }`
`client,r`	`{ account:5000, name:"Jones" }`

演習 2:wordlen 内のバグの探索

さて，WORDCNT に戻って関数 *wordlen* のどこがおかしいのか見つけ出しましょう。

ひたすらコードを調べて，どこがおかしいのか発見できるかどうか見てみるのもよいやり方の1つです。実際，おおよその位置がわかっていれば，それがバグを見つける最も速い方法です。しかし，今回はその誘惑をこらえ，ここでの演習に従ってください。

もし，まだ演習1で開始したデバッグセションにいるのであれば，**Run/Program reset** を選択して（あるいはホットキー ***CTRL-F2*** を押して）今までの操作をキャンセルしてください。これによって Turbo C は，WORDCNT が割り当てていたすべてのメモリを解放し，オープンされているファイルをクローズして，WORDCNT の現在の実行を終わらせます。

一方，統合環境から出てしまったり，演習1を終えて他のプログラムを実行するために統合環境を使用してしまった場合は，統合環境を再開してプロジェクトファイル WORDCNT.PRJ を選択してください。そして，while (fgets(... の文の上にブレークポイントを設定してください。

それでは，**Run/Run** を選択して新しいデバッグセションを開始してください。Turbo C は，再び WORDCNT を走らせるための用意を行ないます。Turbo C はプログラムをブレークポイントで止まるまで走らせます。

ファイル名のプロンプトが表示されたら，WORDCNT.DAT とタイプして**リターン**を押してください。

Run/Step over を選択して WORDCNT を1行ずつ実行し，*wordlen* を呼び出す文まで走らせます。それから，**Run/Trace into** を選択して（あるいはホットキー ***F7*** を使用して），*wordlen* を呼び出す文を走らせるのではなく，デバッガを *wordlen* の中に入り込ませます。

Run/Trace into は **Run/Step over** と同様，プログラムを1行ずつ走らせますが，関数呼び出しを通過せずにその中に入り込みます。この場合，実行バーは *wordlen* の宣言のところに置かれます。

wordlen のアルゴリズムを少し考えましょう。関数は1個の引数（*wordptr* という名前の，ラインバッファ内の語を指すポインタ）を必要とします。まず *wordptr* の値は，ローカル変数 *wordlimit* に代入されます。その後 while ループは，*wordlimit* が語の終わりの空白，または行の終わりのヌル文字を指すようになるまで，*wordlimit* を進めていきます。関数は，*wordlimit* と *wordptr* との差を語の長さとして返します。

代入文から2行だけ実行して，*wordlen* の定義を実行してください。この場合，**Run/Trace into** と **Run/Step over** のどちらを用いてもかまいません。*wordlen* はこれ以上低いレベルの関数を呼び出すことはないので，2つのコマンドは同じ働きをするのです。

ここで *wordlimit* が指している文字列を評価してください。それは，入力ファイルの最初の行であり，それでよいのです。次に while 文を走らせてください。実行バーは次の行に移動し，その行で *wordlimit* はインクリメントされるはずです。ところが実際にはそうではなく，実行バーは後ろの return 文を含む行に移動してしまいます。これが探していたバグなのです。

注意：Evaluate ボックスから出て TC エディタに戻るには，***ESC*** キーを2回押してもかまいませんが，ホットキー ***F10*** を1回押すだけでも戻れることを試してみてください。

次に何が起きるか考えてください。*wordlimit* はインクリメントされていないので，*wordlimit* と *wordptr* との差は0であり，*wordlen* は0を返します。これが，探していたバグです。バグの範囲は，「*wordlen* の中のどこか」から「*wordlen* の while 文の式の中のどこか」に狭められたわけです。ここでもう1度コードを見て，どこがおかしいのかわかるかどうか考えてみる価値はあります。

おかしいところがわからない場合は，式の疑わしい部分を評価して，それらが組み合わされてどのように働くかを考えてみてください。＊ *wordlimit* の値は，ASCII 文字の *T* であることがわかると思います。SPACE は# define で定義された記号なので，＊ *wordlimit* != SPACE は評価できません。しかし，＊ *wordlimit* != ' '（SPACE の値は' '）を評価することは可能で，その値は1（真）です。したがって式全体の値は真になるはずですが，実際には偽になっています。演算子&のどこかがおかしいのです。

実は，&はこの場合誤った演算子なのです。これはＣ言語においてはビット演算子であり，あるオペランドのそれぞれのビットと他のオペランドの対応するビットの論理積をつくる働きを持っています。

＊ *wordlimit* != SPACE は常に0か1なので，＊ *wordlimit* が偶数になれば，＊ *wordlimit* & ＊ *wordlimit* != SPACE は0になってしまいます。この演算子は&&であるべきなのです。この演算子の場合，オペランドが両方とも0でないときに，1という結果を与えます(&と&&の両方の演算子を用いて式全体を評価し，その違いを確認してください)。

(自分でこの問題を解決できなかったとしても，気を落とすことはありません。&と&&，あるいは ¦ と ¦ ¦ を混同することは，初心者のＣプログラマが最も起こしやすい誤りの1つなのですから。自分自身が書いたコードの中で数回この種の誤りを見つけることを経験すれば，すぐわかるようになるはずです)。

バグの修正

バグを修正するのに必要なことは，単に&を&&に変えるだけです。次のデバッグセションでプログラムが壊れても，いま行なった変更が失われないように，直したプログラムをセーブしておきます(***F2*** キーを押してください)。それから，**Run/Run** を選択してください。ソースコードを修正したので，再作成するかどうか聞いてきます。そこで ***Y*** とタイプしてください。そうすると，プログラムのコンパイルとリンクが行なわれます。これで修正したプログラムをデバッグする準備ができたわけです。

さらにバグを探しに行く前に，少し休憩して，今まで学んだことを要約し，これまで出てきたいくつかの点を詳しく見てみましょう。

これまで行なってきたこと

まず，最初のデバッグセションを **Run/Program reset**，あるいはそのホットキー ***CTRL-F2*** でキャンセルしました。それから，ブレークポイントを設定して，WORDCNT を *wordlen* の呼び出しまで走らせ，**Run/Run**（あるいはホットキー ***CTRL-F9***）を選択しました。

Run/Trace into（ホットキー ***F7***）で *wordlen* の呼び出しの中へ入り込みました。これによって *wordlen* をステップ実行し，バグを見つけました。バグを修正し，ソースファイルをセーブして，**Run/Run** で変更を加えたプログラムをデバッグする準備を行ないました。

ブレークポイントについての詳細

最初の演習をやり終えた後 Turbo C から出ていなければ，while (fgets(... に設定されたブレークポイントは，2番目の演習を始めたときにもまだそこに設定されていました。このため WORDCNT は，**Run/Run** でプログラムの終わりまで走らずにブレークポイントで止まったのです。これからわかるように，ブレークポイントはあるデバッグセションから次のデバッグセションへ“引き継がれ”ます。プログラムを編集し，作成し直している場合でもそうです。Turbo C は，各ブレークポイントを適切に上下させて，該当する文のところに保持するわけです。

ブレークポイントは，それらをプログラムファイル中に設定したり，ファイルをセーブしたり，他のファイルを編集したりしても引き継がれます。ブレークポイントが失われるのは次の場合です。

■統合環境から出た場合
■ソースファイル内で，ブレークポイントが設定されている行を削除した場合
■ **Break/watch/Clear all breakpoints** を選択して，ブレークポイントをすべてクリアした場合

しかし，Turbo Cは次の2つの場合にはブレークポイントを追うことができなくなることがあります。

■**ブレークポイントを含むファイルを編集して，そのファイルの編集を中止した（セーブしなかった）場合。** Turbo C は，元のファイルのどこにブレークポイントが設定されていたかを記憶していないので，ブレークポイントを間違った行に表示します。編集したソースファイルを捨てる場合は，すべてのブレークポイントをクリアして（**Break/watch/Clear all breakpoints** を選択して)，必要な場所にセットし直してください。**Break/watch/Clear all breakpoints** は，編集しているソースファイル内だけでなく，プログラム内のすべてのブレークポイントをクリアする点に注意してください。

■**ファイルを編集したのに，コンパイルやリンクをやり直さずに現在のデバッグセションを続けるか，新しいセションを開始した場合。** ブレークポイントは，実際には正しい場所に設定されていますが，ソースファイルがもはや実行可能プログラムと一致していないために，ブレークポイントを示すハイライトは誤った行に現われます（実行バーもまた誤った行に現われます)。

デバッグセッションを続けるか再開しようとするときに，Turbo Cは，Source modified, rebuild? という警告のプロンプトを表示するので，間違ってこのような状況にはまり込むことはないはずです。

ソースファイルをコンパイルする前に，ブレークポイントを実行可能な文を含まない行(たとえばコメント行や空白行)の上に設定することもできます。そのような場合，Turbo Cはプログラムを走らせる直前に，ブレークポイントが実行可能な文を含まないソース行に設定されているという警告を発します。一度ファイルをコンパイルしてしまえば，Turbo Cはどの行が実行可能な文を含んでいるか認識するので，その行にブレークポイントを設定しようとした場合には警告が発せられます。

Break/watch/View next breakpoint を選ぶことにより，カーソルをファイル内の次のブレークポイントに動かすことができます。このコマンドによって，現在のプロジェクトにおける次のブレークポイントにカーソルを動かす点に注意してください。プログラムを走らせたときに次に実行されるブレークポイントではありません。すべてのブレークポイントではなく，その内のいくつかをクリアしたいときには，**Break/watch/View next breakpoint** を使用してプログラムの中のブレークポイントを見まわしておくとよいでしょう。

演習3:プログラムに戻って

では今までの変更でバグが修正されたかどうか見てみることにしましょう。

演習2を終えた後で統合環境から抜けていたり，他のプログラムを走らせたりした場合には，統合環境を再起動して WORDCNT.PRJ をカレントプロジェクトにします。新しくデバッグセッションをスタートするために **Run/Run** を選択してください。*wordlen* を呼び出しているところまで WORDCNT を走らせます。そこまでトレースしたら *wordlen* の中をステップしていきましょう。今度は正しく動いて，値として2が返るはずです。うまくいきましたね!

しかしまだ仕事は終わっていません。このプログラムの大部分はまだテストが済んでおらず，そこにもバグがあるかもしれません。次にテストしなければならないのは **main** の内側の while ループです。最も完全なやり方は，入力の最初の行がすべて処理され制御がループを抜けるまで **main** をステップ実行していくことです。そしてステップごとに正しく処理されているかどうかを確認し，さらに正しい場所で制御がループから抜けているかどうかを確認します。

これは大仕事のように見えるでしょうか?しかしデバッガの Watch ウィンドウは仕事を随分楽にしてくれます。Watch ウィンドウには，式とその式の現在の値とが表示されま

す。デバッガが停止するたびに式の値が計算され，Watch ウィンドウに表示されます。こうして，プログラムの実行中に式の値の変化を**監視**（watch）することができるわけです。

内側の while ループ中で重要なデータ要素――変数 *i*，*wordptr* から始まる文字列，および *wordlencnt* の先頭の要素いくつか――について**監視式**（watch expression）を作成することにしましょう。

Watch ウィンドウは通常，デバッグセッションのあいだ画面の一番下の部分を占めます。これはコンパイル・リンク時に Message ウィンドウがある場所と同じです。初期状態ではウィンドウの高さは1行分で中には何もありません。Watch ウィンドウが表示されていないとすると，Edit ウィンドウが全画面にズームされているのが原因です。***F5*** を押して分割画面の状態に戻せば Watch ウィンドウが表示されます。

Break/**w**atch/**A**dd watch を選択してください（またはホットキーの ***CTRL-F7*** を押してください）。デバッガはウィンドウを開いて監視式の入力を要求します。**D**ebug/**E**valuate と同じように，**B**reak/**w**atch/**A**dd watch も Edit ウィンドウのカーソル位置の単語をデフォルトの式とします。このデフォルトの式が *i* でなかった場合には，*i* をタイプして**リターン**を押してください。デバッガは *i* という式とその現在の値を Watch ウィンドウに追加します。

以上の手続きを，*wordptr* および *wordlencnt*[1]から *wordlencnt*[4]までの5つの式に対して繰り返してください。監視式を追加するごとに，Watch ウィンドウはそれに応じてひろがっていきます。それでは，WORDCNT が入力の最初の行を全部処理するまで，while ループの中をステップ実行して行きましょう。処理が進むにつれて，WORDCNT が *wordptr* を一度に1語ずつ進め，*wordlencnt* の対応する要素をインクリメントしていくのを見ることができます。*wordptr* が行末に達し，実行バーが内側の while ループを抜け出たところでストップしてください。このループは正しく動いているようです。

監視式の編集と削除

Watch ウィンドウ内の監視式は，追加だけでなく編集や削除も可能です。

監視式を編集したり削除したりするには，まず Watch ウィンドウをアクティブにします。メニューシステムの中にいる場合は，まず ***F10*** でそこから抜け出てください。***F6*** を押せば，Edit ウィンドウから Watch ウィンドウに切り換わります。Watch エディタは編集や削除を行なおうとする式をハイライト表示します。ハイライト表示を移動させるには ***HOME***，***SHIFT-HOME***（***End***）および上下の矢印キーを使います。

まず監視式の編集からやってみましょう。ハイライト表示を *wordlencnt*[4]に動かして **B**reak/**w**atch/**E**dit watch を選択します（または**リターン**キーを押します）。するとデバッガはその監視式の入ったウィンドウを開いて式の編集を要求します。

配列の添字を4から6に変えて**リターン**キーを押してください。デバッガはWatchウィンドウ内の監視式を変更し，新しい式の値を表示します。

今度は*wordlencnt*[3]という式をハイライトさせて，Break/watch/Delete watchを選択してください（あるいは***DEL***または***CTRL-Y***を押してください）。デバッガは指定された監視式を削除します。

F6を押してEditウィンドウをアクティブにします。Watchウィンドウがアクティブでない場合は，ハイライト表示されていたことを示すダイヤのマークが監視式の頭に付きます。***F6***をもう一度押してWatchウィンドウを再びアクティブにしましょう。

Break/watch/Remove all watchesを選択すると，すべての監視式を一度に削除することができます（このときWatchウィンドウをアクティブにする必要はありません）。ではすべての監視式を削除してみましょう。Watchウィンドウは元の何もない状態に戻ります。

もう一度***F6***を押してEditウィンドウをアクティブにしてください。***F6***を押すごとに，アクティブウィンドウはEditウィンドウからWatchウィンドウへ，あるいはその逆へと切り換わります。

ウィンドウの拡大と切り換え

デバッガにおけるウィンドウの拡大や切り換えのルールは，これまで説明してきたエディタやコンパイラ・リンカの場合のルールの延長線上にあります。

デバッグセッションの間，画面は通常EditウィンドウとWatchウィンドウとに分割されています。これはコンパイル・リンク時に，画面がEditウィンドウとMessageウィンドウとに分割されているのと同じです。

この表示されている2つのウィンドウ――EditウィンドウとMessageウィンドウ，あるいはEditウィンドウとWatchウィンドウ――を切り換えるには***F6***を押します。

アクティブなウィンドウを全画面に拡大するには***F5***を押します。アクティブウィンドウが拡大表示されているときには，***F5***を押すと画面は分割表示に戻ります。では，EditウィンドウおよびWatchウィンドウについてこの機能を試してみてください。

実行画面を表示させるには，Run/User screenを選択するか***GRPH-F5***を押します。ではこれを試してみてください。どれかキーを押せば元の画面に戻ります。

GRPH-F6はウィンドウの内容を変更するのに使用します。

■ Editウィンドウがアクティブな場合には，現在のファイルの直前にロードされていたファイルが再ロードされます。

■ Watch ウィンドウや Message ウィンドウがアクティブな場合には，***GRPH-F6*** で Watch エディタあるいは Message 追跡ウィンドウとの間で切り換えが行なわれます。

監視式のスクロール

Watch ウィンドウに式を追加していくと，画面の約半分のところまでウィンドウは広がっていきます。そこでさらに式を追加すると，スクロールしてウィンドウの一部が消えてしまいます。***ROLLDOWN***（***PgUp***），***ROLLUP***（***PgDn***）および上下の矢印キーを使ってスクロールさせれば，この消えてしまった式を再び表示させることができます。

1つの監視式が大きすぎてウィンドウに入りきらないときには，***HOME***，***SHIFT-HOME***（***End***），左右の矢印キーを使ってスクロールさせると，式の先頭や最後を見ることができます（Debug/Evaluate で長い式や値をスクロールさせるときと同じです）。

演習 4:プリントループのデバッグ

ここまでで，入力ファイルの読み込みと処理を行なう while ループ全体についてのデバッグが済みました。このループには，バグの隠れていそうな場所がまだ少し残っています。しかしそうした部分をチェックするには，WORDCNT に空行などの特殊な入力を与えてやる必要があります。そこで，そういったバグを探すのは後回しにして結果をプリントする for ループを見てみることにしましょう。

まず for ループが始まる行（117行目あたりです）まで WORDCNT を走らせます。これは，ご存じのようにカーソルをその行に移してブレークポイントを設定し，**Run/Run** を選択すればできますが，今度は別の方法を使ってみましょう。特定の場所で1回だけ停止させたい場合にはこちらの方が便利です。

カーソルを for ループの先頭に置いて，**Run/Go to cursor** を選択します（ホットキー ***F4*** を使うこともできます）。デバッガは WORDCNT を実行し，カーソルのある行でストップします（その前にブレークポイントがあった場合にはそこでストップしますが，プログラムのその部分にはブレークポイントは設定されていません）。

文を1つ実行して，実行バーを for ループの中の **printf** に動かします。for 文の行にはブレークポイントのハイライト表示がないことに注意してください。**Run/Go to cursor** はその場だけの処理です。永久的なブレークポイントを設定するものではありません。**Run/Trace　into** で for ループの中をステップ実行していきましょう。**Run/Trace　into** では **printf** の中はトレースされないことに注意してください。これは **printf** のソースファイルではデバッグ情報を含んだ形でコンパイルするような定義がされていないためです。デバ

ッガはこうした関数を実行することはできますが，その中をトレースすることはできません（ただし今回の場合にはトレースできない理由は他にもあります。**printf**のソースファイル自体がディスクにないことです)。

printfの呼び出しを含む行を実行するたびに，デバッガは実行画面に切り換わります。これによって**printf**は，正しい場所で出力を行なうことができます。デバッガは関数を含んだ文を実行するごとに実行画面を表示します。これは，その関数が実際に画面に出力を行なうのかそうでないのかを知るすべがないからです。

実行画面の表示される短い時間では，プログラムの出力を読むことはおそらく不可能でしょう。そこで，WORDCNTが画面に何をしているかを見るときには，Run/User screenあるいは***GRPH-F5***を使って実行画面を再表示させます。

注意：逆にプログラムの出力がEditウィンドウのソースコードに上書きしてしまうのを避けるには，Debug/Refresh displayを選択して画面を書き直します。そして，Debug/Display swappingオプションがSmartかAlwaysになっているのを確かめてください（このオプションについての詳細は第5章を参照してください)。

これで演習4を終わります。forループにバグがあるかどうかは自分で確かめてみてください。そしてバグがあった場合はどうやって修正するかも考えてみてください。

注意：WORDCNTにはまだいくつかバグがあると言ったはずです。あなたは見つけることができたでしょうか?これらのバグはみんな，この章の後半にある「**効率的なソフトウェアテストの指針**」の節で話題にしているものばかりです。

演習5:大きなプログラムでの作業

デバッガには，大きなソースファイルや複数のファイルからなるプログラムを扱う際に役立つ機能がいくつか用意されています。今度はそういった機能について説明します。

これから説明する機能の1つを使うには，Options/Compiler/Code generation/Standard stack frameオプションをOnにしてプログラムをコンパイルする必要があります。このオプションを確かめてみてください。オプションがOffになっていたらOnにして，EditウィンドウにWORDCNTがロードされているのを確かめてから***GRPH-F9***を押して再コンパイルしてください。

では，Run/Step overを選択するかホットキー***F8***を押してデバッグを開始しましょう。

関数の定義を探す

Debug/Find function は，Edit ウィンドウをスクロールして関数の定義を表示します。このコマンドによって，Debug/Source debugging および O/C/C/OBJ debug information が On の状態でコンパイルしたプログラムはすべて検索することができます。

Debug/Find function は，プログラムのある部分を実行中にバグを発見したが，その修正には別の部分を変更する必要があるといった場合に役に立ちます。また，関数の動作を思い出すために，その関数のコードやコメントを見たいといった場合にも有用です。

このコマンドを試すには，カーソルを **main** 内の *wordlen* を呼び出している問題の行に持っていきます。**Run/Go to cursor** を選び，次にカーソルを *wordlen* という名前に合わせて **Debug/Find function** を選択します。デバッガはウィンドウを開いて関数名の入力を要求してきます。見つけ出したい関数は *wordlen* なので，*wordlen* とタイプして**リターン**を押します。するとデバッガは，Edit ウィンドウに *wordlen* の定義を表示します。

コールスタック

何レベルもの関数呼び出しを行なっているプログラムをデバッグする場合には，**コールスタック**を見る必要の生じる場合がでてくるでしょう。このコールスタックによって，現在の実行位置に至るまでに，プログラムがどんな関数をどんな順序で呼び出しているかがわかります。この機能を使用するには，Option/Compiler/Code generation/Standard stack frame オプションを On にしておく必要があります。

wordlen の中までトレースしてから，**Debug/Call stack** を選択するか，対応するホットキー ***CTRL-F3*** を押してください。デバッガはポップアップウィンドウにコールスタックを表示します。現在実行中の関数はスタックの一番上に表示され，**main** が一番下に表示されます。コールスタックにはプログラムが呼び出している関数名だけでなく引数の値も表示されます。

コールスタックウィンドウのハイライト表示を望みの関数に動かして**リターン**を押せば，その関数の実行中の行を表示させることができます。たとえば **main** の現在実行中の行を表示するには，ハイライト表示をコールスタックウィンドウの **main** に持っていって**リターン**を押します。するとデバッガは Edit ウィンドウをスクロールさせて，**main** の *wordlen* をコールしている部分を表示します。コールスタックに他の関数のエントリがある場合には，**Debug/Call stack** をもう一度選び，続いてコールスタックのその関数のエントリを選択することで別の関数の実行中の行を表示させることができます。

実行中の位置に戻る

Debug/Call stack オプションを使うと，プログラムの他の部分を見た後で実行バーの位置に戻ることもできます。Edit ウィンドウを実行バーのある位置までスクロールさせるには，コールスタックの一番上の関数を選択するだけでいいのです。試してみてください。

これでデバッガに関する練習は終了です。

複数のソースファイルについて

デバッガのコマンドはすべて，複数のソースファイルからなるプログラムでも利用できます。たとえば **Debug/Find function** で，Edit ウィンドウに表示されているのとは別のソースファイルで定義されている関数を検索した場合には，デバッガは適切なファイルをウィンドウにロードします。現在ウィンドウ内にあるファイルに変更が加えられていた場合には，デバッガは新しいファイルをロードする前に現在のファイルをセーブするかどうか聞いてきます。

同様に（**Debug/Call stack** を使って）コールスタック上の関数を選択した場合にも，デバッガは実行位置（現在実行中の関数の次に実行される行）を含むソースファイルを Edit ウィンドウに再ロードします。それ以外のファイルに修正が加えられていた場合には，デバッガは元のファイルを再ロードする前に修正部分をセーブするかどうかを尋ねてきます。プログラムが数多くのソースファイルからなっている場合は，一度にその一部分だけをデバッグするのが賢明です。プログラムに修正を加えるのを一度に数箇所だけに絞れば，仕事を管理するのがずっと楽になります。

デバッガコマンドとホットキーの概要

ここまでは統合デバッガの最もよく使われるコマンドについて説明してきました。デバッガに慣れてきたら勉強したくなるようなコマンドはこれ以外にもたくさんあります。こうしたコマンドについては下の2つの表を参照してください。

デバッガコマンドやそれ以外のメニューコマンドの多くは，ホットキーやキーの組み合わせで起動することができます。しかしあまり細かく説明しても混乱を招くだけでしょうから，これまでは非常に重要なものだけを説明してきました。**表 4.3** にあるのがこれまでに学んだデバッガコマンドについてのすべてのホットキーです。

表 4.3　デバッガコマンドとホットキー

ホットキー	メニューコマンド / 説明
F4	Run/Go to cursor プログラムを実行してカーソルのある行で停止します。デバッグセッションを開始します。
CTRL-F2	Run/Program reset 現在のデバッグセッションをキャンセルして，割り当てられたメモリを解放し，ファイルをクローズします。デバッグセッション内でのみ有効です。
F7	Run/Trace into 現在の関数の次の文を実行します。**Debug/Source debugging** と **O/C/C/OBJ** debug information トグルが On の状態でコンパイルされた関数の呼び出しの場合には，その関数の中もトレースします。デバッグセッションを開始します。
F8	Run/Step over 現在の関数の次の文を実行します。呼び出される関数の内部はトレースしません。デバッグセッションを開始します。

O/C/C/Standard stack frame

Option/Compiler/Code generation/Standard stack frame トグルを切り換えます。Debug/Call stack オプションを正しく機能させるには，プログラムをコンパイルする際にこのトグルを On にしておかなければいけません。

O/C/C/OBJ debug information

O/C/C/OBJ debug information トグルを切り換えます。デバッグができるのはこのトグルを On に設定してコンパイル・リンクされたソースファイルだけです。

CTRL-F4 Debug/Evaluate

C の式を評価します。また変数の値を変更することもできます。

Debug/Find function

関数の定義を検索し，Edit ウィンドウに表示します。デバッグセッション内でのみ有効です。

CTRL-F3 Debug/Call stack

コールスタックを表示します。コールスタックから関数名を選択することで，その関数の現在実行中の行を表示させることができます。デバッグセッション内でのみ有効です。

Debug/Source debugging

デバッグを可能にするかどうかを制御します。これが On に設定されているときは統合デバッガとスタンドアロンのデバッガが両方とも使用できます。Standalone に設定されているときには，プログラムのデバッグにはスタンドアロンのデバッガしか使用できません。ただし TC でもプログラムの実行は可能です。None に設定されている場合には.EXE ファイルにデバッグ情報が書き込まれないので，どちらのデバッガを使ってもプログラムのデバッグは行なえません。

CTRL-F7 Break/watch/Add watch　　監視式を1つ追加します。

Break/watch/Delete watch　　監視式を1つ削除します。

Break/watch/Edit watch　　監視式の編集を行ないます。

Break/watch/Remove all watches　　監視式をすべて削除します。

CTRL-F8　Break/watch/Toggle breakpoint

カーソルのある行のブレークポイントを設定/解除します。

Break/watch/Clear breakpoint

プログラム中のすべてのブレークポイントを解除します。

Break/watch/View next breakpoint

次のブレークポイントを表示します。

表 4.4 にはこれ以外のメニューコマンドで，統合デバッガでよく使用されるものを示します(これ以外のメニューコマンドのホットキーについては第5章を参照してください)。

表 4.4　デバッガで使用されるメニューコマンドとホットキー

ホットキー	メニューコマンド / 説明
F5	アクティブウィンドウを全画面モードに拡大したり分割画面モードに戻したりします。
GRPH-F5	ディスプレイをユーザ画面に切り換えます。どれかキーを押すと統合環境に戻ります。
F6	Edit ウィンドウと，Watch ウィンドウまたは Message ウィンドウの間でアクティブウィンドウを切り換えます。
GRPH-F6	Edit ウィンドウがアクティブな場合には，その前にエディタにロードされていたファイルに切り換わります。下側のウィンドウがアクティブな場合は，Watch ウィンドウと Message ウィンドウの切り換えを行ないます。
CTRL-F9	**R**un/**R**un デバッガ中で，あるいはデバッガなしでプログラムを実行します。必要な場合にはソースファイルのコンパイルやプログラムのリンクも行ないます。プログラムが **D**ebug/**S**ource debugging と **O**/**C**/**C**/**OBJ** debug information が On の状態でコンパイル・リンクされている場合には，ブレークポイントまであるいはプログラムの終わりまで実行します。
	Project/**R**emove message　　Message ウィンドウの内容を消去します。

効率的なソフトウェアテストの指針

ソフトウェアのテストには，デバッガの使用法を知ること以外にも様々なポイントがあります。プログラムが間違った動作をしているかどうか，またその原因はどこにあるのかを知ることは，プログラム開発の上で最も挑戦的な部分です。

この章の残りの部分では，そうしたプログラムテストの作業を容易にするためのテクニックについて述べていきます。

標準的アプローチの展開

ソフトウェアのテストに対する標準的なアプローチを展開していきましょう。すなわち経験から得られたステップからなるチェックリストによって，プログラムの信頼性を高めていくという方法です。

プログラムのテストには唯一の正解というものは存在しません。あなたのチェックリストは，書いているプログラムのタイプ，あなたのプログラマとしての長所と短所，あなた自身の流儀といったものによって変わってきます。しかし次に示すチェックリストを出発点として提供することはできるでしょう。

- ■プログラムに単純でしかも簡単すぎないような入力を与えます。コードをトレースし，**Debug/Evaluate** と監視式を充分に使ってデータ項目の値をチェックします。見つかったバグは修正しますが，一度に修正するのは1箇所ないしは数箇所だけにしておきます。
- ■前のステップでテストできなかった部分を実行するように，プログラムに別のデータセットを与えます。
- ■プログラムのすべての文をテストします。バグがあるとは疑ってもみなかった所でバグが発見させることもあるでしょう。たとえば WORDCNT の場合には，すべての文をテストすると，**else** の後ろに怪しげなセミコロンが見つかります。これによってプログラムが MAXWORDLEN 文字以上の単語に出くわしたときにとんでもないことが起こります（このようなセミコロンは通常 Edit ウィンドウの右端より先にあります。このため，こういうとき以外ではなかなか気がつかないと思います)。
- ■2回以上テストしなくてはいけない文や式に注意します。たとえば次の場合では，

```
if ( strcmp(a,b) ) . . .
```

strcmp は，0（*a* と *b* が等しい場合），-1（*a* が *b* より小さい場合），+1（*a* が *b* より大きい場合）の3種類の値をとり得ます。このことは，**strcmp** がいずれの場合にも正しく動いていることを確かめるには，この文を3種類の入力値でテストすべきであることを示しています。

```
x = (x>0) ? func(x) : 0 ;
```

この文には"暗黙の if"が含まれていて，2つの異なった結果を取り得ます。

■**境界条件**には特別の注意を払ってください。境界条件とは，プログラムがループから脱出する条件や配列をいっぱいにする場合などのことです。境界条件の取り扱いが正しくなかったためにバグが現われるのはよくあることです。

WORDCNT には境界条件に関するバグの実例が2つ含まれています。1つは，最後の for ループは，*wordlencnt* の要素のうち，MAXWORDLEN 文字の長さの単語を表わすものをプリントできません。もう1つは，*wordlencnt* の定義では割り当てられた各要素が小さすぎます。これでは要素が存在できません。

■デバッガから離れて，プログラム全体が正しく動くかどうかをテストしましょう。プログラムを他の人が使うことになっていて，その人はプログラムは当然正しく動くものと思っている場合には，プログラムが出会うかもしれないあらゆる種類のエラーに対する反応をテストしてください。ほとんどのエラーをうまく処理できるプログラムは頑丈だということができます。

■他の人がそのプログラムを使用する予定ならば，プログラムをテストしてくれる人を少なくとも1人用意しましょう。その能力や要求が，プログラムが想定している典型的なユーザになるべく近い人を選びます。ただし，不明確なバグを見つけだして正確にレポートしてくれるような忍耐力と情熱が必要です。あなたのプログラムがプログラマに使ってもらうものでない限り，プログラマをテスターに選ぶのはやめましょう。

変更箇所は完全にテストしよう

プログラムの変更を行なった場合には，それによって影響を受ける部分は完全に再テストを行ないましょう。変更していない部分でも，変更によって影響を受ける部分は再テストを行なうべきです。

プログラムが複雑な場合には，実行したテストの記録をとっておくようにしましょう。プログラムを変更した場合には，変更によって結果が影響を受けそうなテストはやり直すわけですが，その際にこの記録が役に立ちます。テストに特別な入力ファイルがあるときには，そのファイルも取っておくようにしましょう。

用心深い設計

用心深く車を運転すれば事故が避けられるのと同じように，プログラムを用心深く設計することによってバグを避けることができます。

コードはインデント（段下げ）を統一し，気前よくコメントを入れ，説明的な変数名を使うようにしてわかりやすく書きましょう。

コードは単純にしましょう。こみいった処理は数行の複雑な文ではなく，簡単な文をたくさん使って表わすようにしましょう。Turbo C の最適化機能によって，コードは充分効率的になりますし，短い文にすればプログラムのデバッグや理解や変更もずっと容易になります。

プログラムは，目的が単純できちんとしている関数で構成するように心掛けましょう。こうすればテストの条件設定や結果の解析が容易になります。またプログラムを読んだり変更したりするのも容易になります。

各関数が要求するデータ要素および各関数で変更されるデータ要素はなるべく少なくしましょう。このことでもテストの条件設定や結果の解析，プログラムを読んだり変更したりすることが容易になります。またこうすることで，間違った動作をするプログラムによるダメージが小さくなる傾向があります。そのため，1回のデバッグセッションでプログラムを何回も実行することができます。このように設計されたプログラムは，結合がゆるやかであると呼ばれます。

プログラムを書く際には効率をぎりぎりまで追い求めてはいけません。コードを可能なかぎり効率的に書こうとすると，そのプログラムは読んだりデバッグしたりするのが難しくなる傾向があります。できあがった後でプログラムが遅すぎると判明した場合には，今度はどの部分がスピードアップする価値があるのか，どうすれば最も速くなるのかを決めることになります。

プログラム中で2通り以上の方法で使用される関数，および別のプログラムで再利用される可能性のある関数を書く場合には充分な注意を払いましょう。一般化された関数を1つ作成しデバッグすることは通常，特殊な関数を2つ以上書くよりも簡単です。

ボトムアップのデバッグ

可能なかぎり，最初はプログラムの最も低いレベルの関数（他の関数をコールしない関数）のデバッグに専念しましょう。その後，作業が **main** まで順に上がっていくようにします。そうすることにより，プログラムの別の部分でその関数をコールしているときには，信頼できる基盤となる関数が得られることになります。

同種のバグを探す

バグが見つかった場合には，同じ種類のバグが他にもないか，プログラムの同じような部分にバグがないかを注意深く調べましょう。たとえば次のような関数呼び出しが見つかったとします。

```
fp = fopen("rb",filename);
```

これは次の形でなければなりません。

```
fp = fopen(filename,"rb");
```

こういう場合には，これ以外の **fopen** や類似の関数の呼び出しで同じエラーがないかどうかをチェックしてください。

インラインアセンブリコードのデバッグ

統合開発環境をボーランド社の TASM と組み合わせて使っている場合には，他のデバッガを使用しなくてもアセンブラレベルのステップ実行が可能です（ただし充分な機能を持つアセンブラレベルのデバッグには，ボーランド社のスタンドアロンデバッガを入手すべきです）。

アセンブラモジュールが TASM で-zi スイッチを指定してアセンブルされている場合には，TC はアセンブラのソース行とシンボルを認識することができます。アセンブラレベルの関数の中をステップ実行すると，TC はその関数のアセンブリ言語のソースコードを表示します。このアセンブリソースコードに対して，**Debug/Go to cursor**（***F4***），**Debug/Trace into**（***F7***），**Debug/Step over**（***F8***）などの通常の TC のデバッグコマンドを使用することができます。

アセンブリソースコードで定義されているほとんどのシンボルは，評価式や監視式として使用することができます。さらに，疑似レジスタ（_AX，_BX など）や，CPU のフラグレジスタの状態を表わす特殊変数_FLAGS にもアクセスできます。ただし TC は C の開発環境ですから，アセンブラの構造や式をすべて認識できるわけではありません。

第5章

Turbo C 統合開発環境

TC――Turbo C 統合開発環境は，単にコンパイルが速いシステムではありません。速くて効率的な C コンパイラが，学びやすくかつ使いやすい組み込みのエディタ，デバッガ，ユーティリティなどとともに組み込まれているのです。TC では，C のプログラムを走らせるために，別にエディタ，デバッガ，コンパイラ，リンカ，MAKE ユーティリティなどを用意する必要がありません。これらはすべて TC の中に組み込まれているので，TC メイン画面と呼ばれる画面から簡単に呼び出すことができるのです。

この章では……

この章は4つにわかれており，**第 I 部「TC を使う」**，**第 II 部「メニューコマンド」**，**第 III 部「コンフィギュレーションファイルとピックファイル」**，**第 IV 部「追加された機能およびエディタコマンド」** となっています。

第 I 部「**TC を使う**」では以下の点について説明します。

- ■ TC コマンドラインスイッチとホットキーの紹介
- ■ Turbo C メイン画面の各要素
- ■ Turbo C メインメニューの選択の手順
- ■ Edit ウィンドウへの入り方，エディタの使い方（エディタのコマンドについては，付録 A「Turbo C **エディタ**」で詳しく説明されています）
- ■ TC 統合デバッガの紹介（デバッガの使い方の詳細については第4章「**プログラムのデバッグ**」を参照してください）

第 II 部「**メニューコマンド**」では次のような話題を取り上げます。

- メニューの各項目の働き
- コンパイル時オプションの要約

第 III 部「**コンフィギュレーションファイルとピックファイル**」では，

- コンフィギュレーションファイルとは何か，どのように作成し，TC の中でどのように使用するかについて紹介します。
- TC の中で，ピックファイルをどのように作成し，どのように使うかについて紹介します。

第 IV 部「**追加された機能およびエディタコマンド**」では，

- メニューシステムからは使用できない編集機能を紹介し，
- Turbo C のカスタマイズプログラム TCINST.EXE を使ってユーザ独自のエディタキーをカスタマイズするやり方を説明します。

次に読む場所

ボーランドの統合開発環境に慣れていない方は，第 I 部と第 II 部を先に読んだ方がよいでしょう。SideKick や Turbo Prolog などのメニュー形式のソフトウェアをすでに使っている方は，第 I 部と第 II 部を飛ばして第 III 部および第 IV 部へ進んでもかまいません。

ヘルプ情報を得るには

他のボーランド社の製品と同様に，Turbo C は現在の状況に即したヘルプ情報を，キーを1つ押すだけで得ることができます。ヘルプ情報は，Turbo C のメニューのどの地点からでも見ることができます。

ヘルプシステムを呼び出すには ***F1***（または ***HELP*** キー）を押します。Help ウィンドウには，現在位置づけられている項目の各機能についての説明が表示されます。どのヘルプ画面にもキーワード（ハイライト表示されている）が1つ存在し，それを選択するとより詳細な情報が与えられるようになっています。矢印キーを使って希望するキーワードにハイラ

イトバーを動かして**リターン**キーを押すと，その項目に関する詳細なヘルプ情報が得られます。***HOME*** と ***SHIFT-HOME*** キー（***End***）を使うと，それぞれ画面上の最初または最後のキーワードにハイライトを移動することができます。

前に見ていたヘルプ画面に戻りたいときには，ヘルプシステムの中にいても出てしまっていても ***GRPH-F1*** を押します。最大20個前のヘルプ画面までさかのぼることができます。

ヘルプの索引を表示するには，ヘルプシステムの中でもう一度 ***F1*** を押します。

TC エディタの中にいるときには，さまざまなライブラリ関数に関するヘルプが必要になることがあります。こんなときには情報が欲しい関数名（たとえば **printf**）のところにカーソルを置いて ***CTRL-F1*** を押せば，その関数についての情報を含んでいるヘルプ画面を表示させることができます。

ヘルプシステムから出てメニューの選択に戻るには，***ESC***（または次の節で述べるホットキーのいずれか）を押します。

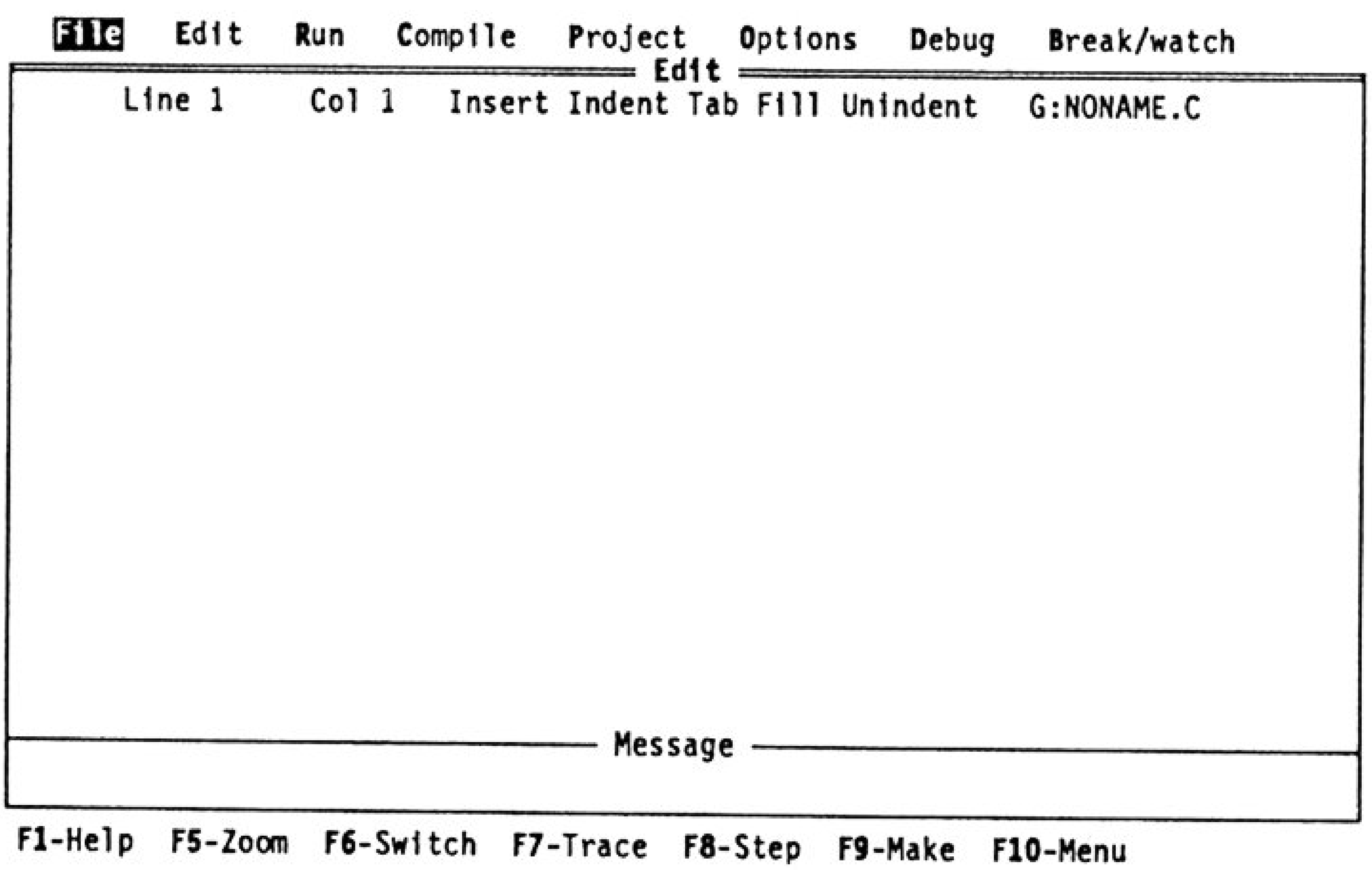
第I部：TCを使う

Turbo C統合開発環境（TC）をロードするには，MS-DOSのプロンプトに対してtcとタイプしてに**リターン**を押します。

最初に現われるスタートアップ画面には，TCメイン画面と，製品バージョン情報を含むボックスが表示されます。どれかキーを押すとバージョン情報は消えますが（***SHIFT-F10***を押せばいつでもこの情報が表示されます），メイン画面はそのままです（**図5.1**）。

TCメイン画面を少し詳しく見てみましょう。この画面は4つの部分——メインメニュー，Editウィンドウ，Messageウィンドウ，クイックリファレンス行から構成されていることがわかります。

図5.1　TCメイン画面



TC コマンドラインスイッチ

Turbo C 統合開発環境（TC）は，以下のコマンドラインスイッチを受け入れます。

■/C スイッチは，コンフィギュレーションファイルをロードすることを指示します。tc コマンドのあとに，/c とコンフィギュレーションファイルの名前を入力してください（/c とコンフィギュレーションファイル名の間にはスペースを入れてはいけません）。

```
tc /cmyconfig.tc
```

（コンフィギュレーションファイルについてはこの章の第 III 部で詳しく採り上げます）。

■/b スイッチは，TC にプロジェクト中のすべてのファイルを再コンパイルしてリンクすることを指示します。コンパイラは標準出力装置にコンパイラメッセージをプリントして DOS に戻ります。このスイッチを使うと，バッチファイルから TC を起動することができるので，プロジェクトを自動的にプロジェクトの作成（Build）を行なうことができます。TC は作成の前に，デフォルトのコンフィギュレーションファイルまたは/c スイッチで指定されたコンフィギュレーションファイルをロードします。作成する.EXE ファイルがなにかは，コンフィギュレーションファイルあるいはピックファイルの内容から，(1)プロジェクトファイル，(2)プライマリファイル，(3) TC エディタにロードされているファイル，の順で決定されます。tc コマンドのあとに，/b スイッチのみ，あるいは/c とコンフィギュレーションファイル名に続いて/b スイッチを入力してください。

```
tc /cmyconfig.tc /b
tc /b
```

ロードされるコンフィギュレーションファイルにプロジェクトファイルまたはプライマリファイルが設定されていない場合には，コンパイルおよびリンクされるプログラムの名前をコマンドライン上で指定することができます。tc コマンドのあとにプログラム名をタイプし，/b をつけてください。

```
tc myprog /b
```

■/m スイッチは，Build ではなく Make を行ないます(つまり，新しくなっているソースファイルだけが再コンパイルされ，リンクが行なわれます)。/b を/m に置き換える以外は，/b スイッチの説明にしたがってください。

■/d スイッチは，適切なハードウェアが検出された場合には，TC をデュアルモニタモードで動かします。適切なハードウェアが検出できない場合にはこのスイッチは無視されます。デュアルモニタモードは，プログラムの実行時やデバッグ時，および DOS のシェルを呼び出す（**File/OS shell**）ときに使用されます。

注意：デュアルモニタモードは IBM PC でのみサポートされています。PC-9801シリーズでは，ハードウェアの設計上この機能は実現できないため，このスイッチは常に無視されます。

TC を起動するコマンドラインで/d スイッチを指定した場合，必要なハードウェア（たとえばモノクロカードとカラーカード）が存在していれば，デュアルモニタモードが有効です。使用しているシステムが2つのモニタを備えていれば，DOS はその一方をアクティブなモニタとして扱います。DOS がどちらをアクティブとするかは，DOS の MODE コマンドによって切り換えてください（たとえば MODE CO80，MODE MONO など）。デュアルモニタモードでは，通常の TC の画面はアクティブでないモニタに表示され，プログラムの出力はアクティブモニタに送られます。したがって，一方のモニタで DOS プロンプトから tc /d とタイプした場合，TC はもう一方のモニタに現われます。プログラムを特定のモニタでテストしたいときには，一度 TC を終了して，テストの対象となるモニタをアクティブに切り換えてから，tc /d によってもう一度 TC を起動してください。プログラムの出力は，tc コマンドを入力したモニタに対して行なわれます。

警告：

・DOS のシェルの中にいる（**File/OS shell**）ときには，（DOS の MODE コマンドなどによって）アクティブモニタを切り換えてはなりません。

・アクティブでないモニタのビデオカード上のポートを直接アクセスするプログラムはサポートされておらず，予期できない結果を引き起こすことがあります。

・プログラム側でデュアルモニタを使う場合，そのプログラムの実行およびでバッグを行なう際には，TC のデュアルモニタスイッチ（/d）は使ってはなりません。

TCの中を動き回る

Turbo C統合開発環境に慣れ親しむために，TCの中を動き回り必要な項目にすばやく達するための基本的な操作を以下に示しておきます。

Turbo Cの中ならどこからでも：

- ***F1*** を押すと現在の位置に関する情報が得られます(実行やコンパイルに関するヘルプ情報など)。
- ***F5*** を押すと，アクティブウィンドウのズームイン/ズームアウト(拡大/縮小)を行なうことができます。
- ***F6*** を押すと，ウィンドウが切り換わります。
- ***GRPH-F6*** はウィンドウの内容を切り換えます(Messageウィンドウと Watchウィンドウの切り換え，あるいは現在のファイルとその前のファイルの切り換え)。
- ***F10*** は，メニューとアクティブウィンドウの間を行き来するトグルになっています。
- ***GRPH*** キーを押しながらメインメニューの先頭の文字(***F,E,R,C,P,O,D***)を押すと，そのコマンドが実行されます。たとえば，Turbo Cシステムのどこにいても，***GRPH-E*** を押すとEditウィンドウに入り，***GRPH-F*** でFileメニューに入ることになります。

メニューの中から：

- メニューの項目を選択するには，ハイライト表示されている大文字をタイプするか，矢印キーを使ってその項目に移動してから**リターン**キーを押します。
- メニューから抜け出るには ***ESC*** を押します。
- メインメニュー，あるいはメインメニューから呼び出されたプルダウンメニューの中で ***F10*** を押すと，即座にアクティブウィンドウに戻ることができます(ウィンドウがアクティブになると，その名前がハイライト表示され，さらにウィンドウの上枠が二重線で表示されます)。
- どんなメニューレベルからでも，***F6*** を押すと，その前にアクティブだったウィンドウに戻ります。
- 左右の矢印キーを使うと，プルダウンメニューの中から他のプルダウンメニューへ移ることができます(ただし移動ができるのは，一番上のレベルのプルダウンメニューにいる場合だけです)。

TC を終了して DOS に戻るには：

File メニューに入って **Quit** を選択します（***Q*** を押すか，ハイライトバーを **Quit** に移動して**リターン**キーを押します)。現在のワークファイルをセーブせずに **Quit** を選択すると，エディタはそのワークファイルをセーブするかどうかを聞いてきてくれます（ホットキー GRPH-X は，**File/Quit** を呼び出すのと同じです)。

TC ホットキー

さまざまなメニューのオプションについて説明していく前に，知っておかなければならない**ホットキー**（短縮コマンド）というものがいくつかあります。ホットキーというのは，なんらかの機能を実行するために設定されているキーのことです。たとえば前にも出てきたように，***GRPH*** キーを押しながらメインメニューのコマンドの頭文字を押すと，指定したオプションメニューに入るか，なんらかの動作を行ないます。***GRPH*-頭文字**コマンドとしては，他に ***GRPH-X*** が1つだけあり，これは **File/Quit** の短縮コマンドになっています。

これらの ***GRPH*-頭文字**コマンドに加えて，TC は特別なユーザ画面ホットキー ***GRPH-F5*** を持っています。このキーを使うと，TC メイン画面とユーザ画面（ユーザプログラムの出力が表示されている画面）の切り換えを行なうことができます。これは **Run** メニューの **User screen** メニューコマンドと同じです。

TC を使用する場合，2つの画面，すなわち TC メイン画面とユーザ画面を見ることになります。TC メイン画面は，プログラムを編集，コンパイル，リンク，デバッグするときに見る画面であり，ユーザ画面は Turbo C の実行可能プログラムを実行するときや，**File/OS shell** コマンドで一時的に DOS に戻ったときに見る画面です。統合デバッガを使用する場合は，この TC 画面とユーザ画面を頻繁に切り換えることになります。TC はユーザ画面の内容をユーザ画面バッファにしまっておき，実行のコマンド（たとえば **Run**，**Trace into**，**Step over**）や **File/OS shell** が選択されるたびにその内容を更新します。このセーブされている画面を見るには，**Run** メニューから **User screen** を選ぶか，***GRPH-F5*** を押します。

注意：（IBM PC）デュアルモニタモードの場合は，ユーザ画面はすでにシステムの2個のモニタのいずれかに表示されているはずです。したがって，**Run/User screen** コマンドおよび ***Alt-F5*** は使用できません。

TC がユーザ画面をクリアする必要があるかどうかは，ビデオモードによって決められます。TC が DOS から呼び出されたとき，あるいは DOS シェルから TC に戻ったときに，

TCはビデオモードとカーソルの形状を記憶します。ビデオモードとカーソルの形状はそれぞれ別個に記憶されており，DOSへ抜け出たり（**File/OS shell**），統合環境を終了した（**File/Quit**）ときに，現在の状態が記憶されている状態と異なっているときには元の状態にリセットされることになっています。ただし，これには1つだけ例外があります。デバッグセッションの中で（プログラムが走っているとき）DOSへ抜けでた場合は，モードとカーソルの形状はユーザプログラムがセットした状態になったままとなります。

表5.1は，TCの中で使用できるすべてのホットキーを示しています。これらのキーを押すと，TC環境のどこにいるかには関係なく，その機能が働くことを憶えておいてください。これにも例外が1つだけあります。対話ボックスが表示されて，ある特定のキーの入力が要求されている場合には，そのキーが入力されるまでホットキーは使用できません。

表5.1　TCホットキー

キー	機能
F1	現在の場所に関するヘルプ情報をヘルプウィンドウに表示します。
F2	エディタで現在処理中のファイルをセーブします。
F3	ファイルをロードします（入力ボックスが現われます）。
F4	カーソルがある行までプログラムを実行します。
F5	アクティブウィンドウをズームイン/ズームアウトします。
F6	アクティブウィンドウを切り換えます。
F7	デバッグモードでプログラムを走らせます（**Trace into**）。
F8	デバッグモードでプログラムを走らせます（**Step over**）。
F9	Makeを実行します。
F10	メニューとアクティブウィンドウを切り換えます。
CTRL-F1	関数に関するヘルプ情報を呼び出します。
CTRL-F2	実行中のプログラムをリセットします。
CTRL-F3	コールスタック（Call stack）を呼び出します。
CTRL-F4	式を評価します。
CTRL-F7	監視式を加えます。
CTRL-F8	ブレークポイントのオン/オフを切り換えます。
CTRL-F9	プログラムを実行します。
GRPH-F1	最後に見たヘルプ画面をもう一度表示します。
GRPH-F3	ロードするファイルを選択します。
GRPH-F5	TCメイン画面とユーザ画面を切り換えます。

GRPH-F6	アクティブウィンドウの内容を切り換えます。
GRPH-F7	1つ前のエラーに移ります。
GRPH-F8	次のエラーに移ります。
GRPH-F9	エディタ内のファイルをコンパイルして.OBJを作ります。
GRPH-B	Break/watchメニューに入ります。
GRPH-C	Compileメニューに入ります。
GRPH-D	Debugメニューに入ります。
GRPH-E	エディタに入ります。
GRPH-F	Fileメニューに入ります。
GRPH-O	Optionsメニューに入ります。
GRPH-P	Projectメニューに入ります。
GRPH-R	Runメニューに入ります。
GRPH-X	TCを終了してMS-DOSに戻ります。
SHIFT-F10	バージョン画面を表示します。

注記：IBM PCでは，***GRPH***はすべて***Alt***になります。

メニューの構造

図5.2は，TCのメインメニューと，その下に続くプルダウンメニューの完全な構造を示しています。TCのメニューの項目は，コマンド，トグル，設定という3つのタイプに大別することができます。

コマンド　なんらかの仕事（実行，コンパイル，オプションの保存など）を行ないます。

トグル　TCの機能のオン/オフを指定したり（Auto dependency，Keep messages，など），複数のオプションの中から希望のものが現われるまでリターンキーを押して選択します（Message tracking，Floating point，など）。

設定　ディレクトリの配置や，ファイルの名前，マクロ定義など，コンパイル時/実行時に必要な情報をコンパイラに指示します。

図5.2 (1) Turbo Cのメニュー構造

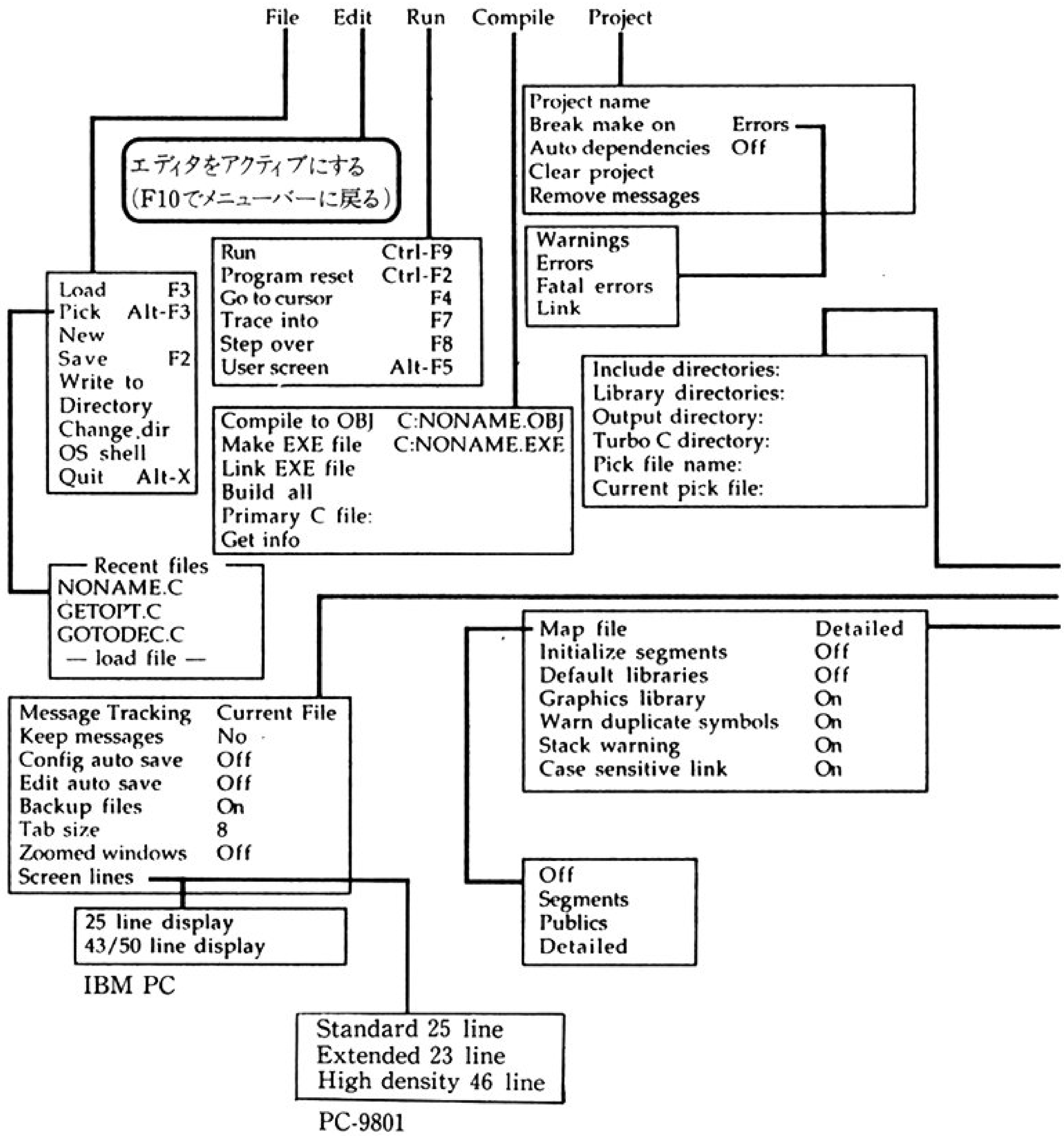
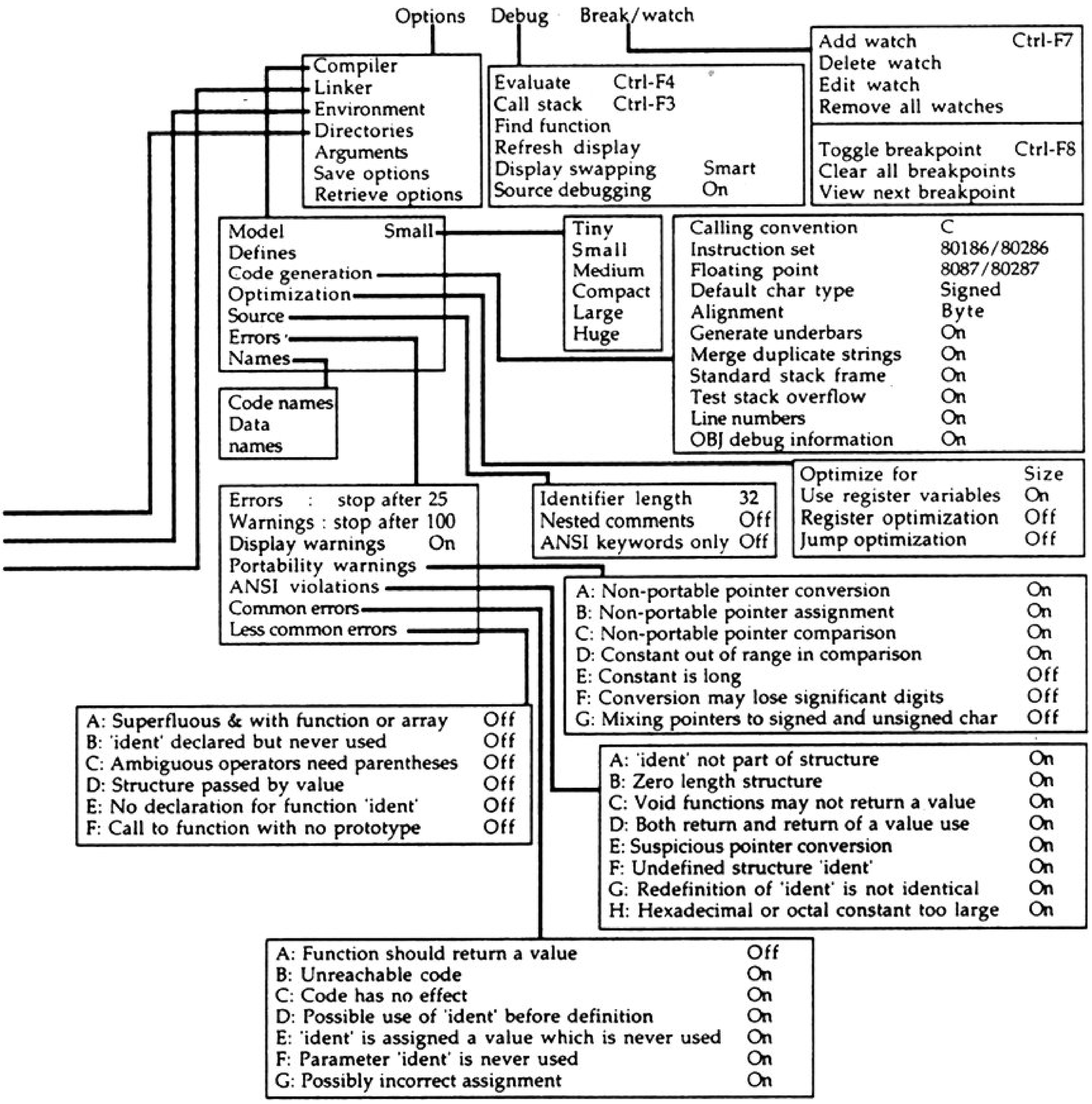


図 5.2 (2) Turbo C のメニュー構造



メニュー名の表現法

このマニュアルでは，メニュー項目は省略名で表記している場合がしばしばあります。省略名は，メインメニューからその項目を選択するまでに入力する文字を並べて表現します。

- ■メインメニューから，コンパイル時のオプションでエラーメッセージに関するメニューを選ぶのは Options/Compiler/Errors ですが，これを O/C/Errors（***OCE*** の順でキーを押す）と表記します。
- ■メインメニューから，インクルードディレクトリの名前を指定するメニューは Options/Directories/Include directories ですが，これを O/D/Include（***ODI*** の順でキーを押す）と表記します。

メインメニュー

メイン画面の一番上の行には，次に示すような Turbo C メインメニューが表示されており，8つの選択肢が用意されています。

```
File   Edit   Run   Compile   Project  Options  Debug  Break/watch
```

File ファイルに関する操作（ロード，セーブ，ピック，新規作成，ディスクへの書き込み）や，ディレクトリの操作（一覧の表示，変更），DOS シェルの起動，TC の終了などを行ないます。

Edit ソースファイルの作成や編集を行ないます。

Run プログラムの実行をコントロールします。また，Debug/Source debugging および O/C/C/OBJ debug information トグルがオンの状態でプログラムをコンパイル・リンクしていれば，このメニューからデバッグセッションを開始することができます。

Compile プログラムのコンパイルや Make を行ない，オブジェクトファイルや実行可能ファイルを作成します。

Project プログラムがどのようなファイルから構成されるかを指定するなど，プロジェクトの管理を行ないます。

Options コンパイラオプション（メモリモデルや，コンパイル時オプション，エラーメッセージ，リンカオプションなど）の選択や，マクロ定義を行ないま

す。また，インクルードファイル，出力ファイル，ライブラリファイルが格納されているディレクトリの指定，コンパイラオプションの保存，コンフィギュレーションファイルからのオプションのロードなども行ないます。

Debug 変数の値をチェックしたり変更したり，関数が定義されている場所を調べたり，プログラムの実行中コールスタックを調べることができます。また，デバッグ情報を組み込んでプログラムをコンパイルするかどうかを選択することができます。

Break/watch 監視式の追加，削除，編集できます。また，ブレークポイントの設定，クリア，次のブレークポイントの表示ができます。

メインメニューのうち **Edit** はコマンドであることに注意してください。**Edit** は単にエディタに入るだけです。他のメニューは，数多くのオプションをもつプルダウンメニューを表示し，オプションによってはさらにサブメニューを表示します。

クイックリファレンス行

ウィンドウあるいはメニューの中にいるときには，画面の最下行に，**クイックリファレンス**行が表示されています。この行を見れば，その時点でのファンクションキーの働きを知ることができます。

最初に TC に入ったとき，デフォルトのクイックリファレンス行は次のようになっています。

```
F1-Help  F5-Zoom  F6-Switch  F7-Trace  F8-Step  F9-Make  F10-Menu
```

GRPH キー（***Alt***）を数秒間押し続けてみてください。クイックリファレンス行の表示は，***GRPH*** キーと組み合せた場合に実行される機能に切り換わって，次のように表示されるはずです。

```
Grp: F1-Last help  F3-Pick  F6-Swap  F7/F8-Prev/Next error  F9-Compile
```

Edit ウィンドウ

この節では，TC メイン画面を構成する要素について述べた後，TC Edit ウィンドウにおける作業の仕方について説明します。

Edit ウィンドウに入るには，まず ***F10*** を押してメインメニューを呼び出し，Edit オプションにハイライトを動かして**リターン**キーを押します（あるいはメインメニューで ***E*** を押します）。メインメニューではなくシステムのどこかにいる場合は ***GRPH-E*** を押します。Edit ウィンドウに入ると，ウィンドウの上枠が二重線になり，その名前（Edit）がハイライト表示されます——これは Edit ウィンドウがアクティブになったことを示しています。

Edit ウィンドウの本体ではソースファイルの編集ができるわけですが，この他に TC 画面には，ステータス行とクイックリファレンス行という情報表示のための2つの行があります。

Edit ウィンドウの最上段のステータス行には，編集中のファイルに関する情報が表示されており，カーソルの現在位置，現在の編集モードを見ることができます。

```
Line   Col   Insert  Indent  Tab  Fill  Unindent  *  C:FILENAME.EXT
```

Line *n* カーソルは *n* 行目にあります。

Col *n* カーソルは *n* 桁目にあります。

Insert 挿入モードはオンになっています。挿入モードは，***INS*** キーまたは ***CTRL-V*** でオン/オフが切り換わるトグルになっています。挿入モードと上書きモード（挿入モード＝オフ）については付録 A で説明しています。

Indent 自動インデント（段下げ）がオンになっています。自動インデントは，***CTRL-O I*** でオン/オフが切り換わるトグルになっています。自動インデントについては付録 A で説明しています。

Tab タブモードはオンになっています。タブモードは，***CTRL-O T*** でオン/オフが切り換わるトグルになっています。

Fill タブモードがオンの場合，エディタは各行の先頭にタブや空白を埋め込みます。このオプションは，***CTRL-O F*** でオン/オフが切り換わるトグルになっています。付録 A を参照してください。

Unindent 行の先頭の非空白文字，あるいは空行にカーソルがある場合に，***BS*** キーを押すと，カーソルは1つ左のインデントのレベルへ戻ります。このオプションは，***CTRL-O U*** でオン/オフが切り換わるトグルになっています。付録 A を参照してください。

***** ファイルが修正されているのに，まだセーブされていないときにアスタリスク（＊）がファイル名の前に表示されます。

C:FILENAME.EXT 編集中のファイルの，ドライブ名（C:），ファイル名（FILENAME）および拡張子（.EXT）です。

TC画面の最下段のクイックリファレンス行は，ホットキーの働きを表示しています。

```
F1-Help  F5-Zoom  F6-Switch  F7-Trace  F8-Step  F9-Make  F10-Menu
```

これらの機能を実行するには，対応するキーを押します。

F1-Help Turbo Cエディタのコマンドのヘルプウィンドウを開きます。

F5-Zoom アクティブウィンドウ(この場合はEditウインドウ)を画面いっぱいに拡大します。***F5***はトグルになっており，もう一度押すと元の分割画面環境に戻ります。

F6-Switch あるアクティブウインドウからもう1つのウインドウに切り換わります(Editと，MessageまたはWatchの間で切り換えを行ないます)。

F7-Trace デバッグモードでプログラムを1行ずつ実行します。関数が呼び出された場合は，その中にも入って1行ずつ実行していきます。

F8-Step デバッグモードでプログラムを1行ずつ実行します。関数が呼び出された場合は，その中をトレースすることはしません。

F9-Make Make(コンパイルとリンク)を行なって.EXEファイルを作成します。

F10-Menu メインメニューに戻るか，あるいはメニューの中からEditウインドウに戻ります。

このエディタは，SideKickのNotepadや，Turbo Pascalのエディタに似たコマンド体系を持っています。こうしたエディタに慣れていない方は，付録Aにエディタコマンドについての詳細な説明が示されているので参照してください。次のページには，頻繁に使用されるコマンドの一覧を示してあります。

エディタ内で挿入モードでプログラムを入力している場合には，行の終了には**リターン**キーを押します(エディタはワードラッピング――行に単語が収まらないときにその単語を次の行の先頭に持っていくこと――を行ないません)。1行の長さは最大248文字まで，ウィンドウの幅は77文字分です(ウィンドウを拡大している場合は79文字)。77桁目を越えて文字を入力すると，1文字入力するごとにウィンドウは横にスクロールしていきます。Editウィンドウのステータス行には，カーソルがファイル内の何行目，何列目にあるかが表示されています。

プログラムの入力がすんだら，***F10***を押してメインメニューに戻ります。メインメニューに移っても入力したファイルは画面上に残っており，もう一度***E***(Edit)または***F10***を押せばEditウィンドウに戻ることができます。

エディタコマンドのクイックガイド

以下は，エディタコマンドの中でもっとも頻繁に使用されるものです。

■カーソルの移動	上下左右の矢印キー（↑，↓，←，→）
■ページ単位のスクロール	***ROLLDOWN***，***ROLLUP***（IBM PC では ***PgUp***，***PgDn***）
■行の削除	***CTRL-Y***
■単語の削除	***CTRL-T***
■ブロックのマーク	***CTRL-K B***（始点），***CTRL-K K***（終点）
■ブロックの移動	***CTRL-K V***
■ブロックのコピー	***CTRL-K C***
■ブロックの削除	***CTRL-K Y***

付録 A にはすべてのエディタコマンドについて詳細な解説があります。

Edit ウィンドウでソースファイルをどう扱うか

特定のファイルを指定せずに Edit ウィンドウを呼び出すと，TC エディタは自動的に，作成するファイルに NONAME.C という名前をつけます。この時点で，エディタのすべてのコマンドが使えるようになり，以下のことが行なえます。

■新しいソースファイルを，NONAME.C または指定した名前で作成する。
■既存のファイルをロードし，修正する。
■今までに編集を行なったことのあるファイルの一覧表からファイルをピック（拾い上げる）し，Edit ウィンドウにロードする。
■ Edit ウィンドウ内のファイルをセーブする。
■エディタ内のファイルを新しい名前でディスクに書き出す。
■コンパイル時のエラーの確認と修正のために，Edit ウィンドウと Message ウィンドウの間を移動する。

ソースファイルの作成/編集中は，まだコンパイルしていないのであれば Message ウィンドウは必要ないので，***F5*** を押して Edit ウィンドウを画面いっぱいに拡大することができます。Edit ウィンドウを元の大きさに戻すには，もう一度 ***F5*** を押します（元の分割画面モードに戻ります）。

新しいソースファイルの作成

新しくファイルを作成するには，次の2つの方法があります。

■メインメニューから **File/New** を選択して**リターン**キーを押します。これで，NONAME.C というファイル名で Edit ウィンドウがオープンされます。

■メインメニューから **File/Load** を選択します。ロードするファイル名の入力をうながす Load File Name ボックスが表示されるので，新しく作成するファイル名を入力して，**リターン**キーを押します(Edit ウィンドウの中からホットキー ***F3*** を押すと，同じように Load File Name ボックスが現われます)。

既存のソースファイルのロード

既存のファイルをロードして編集を行なうには，**File/Load** または **File/Pick** の2つの方法があります。

メインメニューから **File/Load** を選択すると，次のようなことが行なえます。

■編集したいファイルの名前を入力します。パス名も指定できます。

例：　C：¥TURBOC¥TESTFILE.C

■ Load File Name ボックスにマスクを入力して（ワイルドカード＊や?を使う）**リターン**キーを押します。ワイルドカードを使うと，該当するファイル名の他にディレクトリ名も表示されます。＊.＊を入力すると，サブディレクトリも含めてカレントディレクトリ内のすべてのファイルが表示されます。ディレクトリ名には最後に円記号(¥)がつけられます（..¥は1つ上のディレクトリを示します)。ディレクトリを選択すると，そのディレクトリ内のファイルが表示されます。たとえば A:¥＊.C と入力すると，ルートディレクトリにある拡張子が.C のファイル(およびサブディレクトリ名)が表示されます。

上下左右の矢印キー（↑，↓，←，→）を使って，ハイライト表示を選択したいファイルに合わせます。そのあと**リターン**キーを押せば選択したファイルがロードされ，Edit ウィンドウに移ります。

File/Pick を選択するか ***GRPH-F3*** を押しすと，今までにロードしたことのあるファイルをすばやく選択することができます（File/Pick メニューについては，この章の第 II 部

で説明しています)。

直前にロードされていたファイルを再ロードするホットキー ***GRPH-F6*** が追加されています。***GRPH-F6***（ウィンドウ内容の切り換え）は，エディタ内の現在のファイルと，その前にロードされていたファイルを切り換えます。

ソースファイルのセーブ

次の2つの方法で現在の編集ファイルをセーブすることができます。

■システムのどこからでも ***F2*** を押します。

■メインメニューから File/Save を選択します。

ファイルに書き出す

エディタの中のファイルを，新しいファイルに書き出したり，すでにディスク上にあるファイルに上書きすることができます。カレント（デフォルト）ディレクトリに書き出すこともできますし，他のドライブやディレクトリを指定することもできます。

まず，メインメニューで File/Write to を選択します。New Name ボックスが表示されるので，書き出すファイルのフルパス名を，たとえば次のように入力してリターンキーを押します。

```
A:¥DIR¥SUBDIR¥FILENAME.EXT
```

■ A:はドライブ名です（カレントドライブ以外のドライブに書き出すときに指定します)。

■¥DIR¥SUBDIR¥はディレクトリ名です（カレントディレクトリ以外のディレクトリに書き出すときに指定します)。

■ FILENAME.EXT は，出力するファイルの名前と拡張子です。拡張子 (.EXT) を指定しなければ.C とみなされます。拡張子をつけたくなければ，ピリオド(.)のみをつけます。

指定した名前のファイルがすでに存在している場合は，書き出しを行なう前に，上書きしてもよいかどうかをたずねてきます。

ESC を一回押すとアクティブウィンドウ(Edit ウィンドウ)に戻ることができます。***F10*** か ***GRPH-E*** でもエディタに戻れます。

注意：TC エディタに関する包括的な解説については付録 A を参照してください。

Message ウィンドウ

Message ウィンドウは，ソースファイルのコンパイルおよびデバッグを行なうときに，診断メッセージを参照するために使用します。TC には独特なエラー追跡機能があり，コンパイル時に発生した警告およびエラーのメッセージが Message ウィンドウに表示され，同時に Edit ウィンドウには，各メッセージに対応するソースファイル内の箇所がハイライト表示されます（**Option**/**Environment** メニューの **Message tracking** コマンドの設定にも依存します）。

カーソルが Message ウィンドウにあるときには，クイックリファレンス行に表示されるホットキーの機能は次のようになっています。

F1-Help	Turbo C のエラー追跡機能に関するヘルプウィンドウを開きます。
F5-Zoom	Message ウィンドウを画面いっぱいに拡大します。
F6-Switch	Edit ウィンドウをアクティブにします。
F7-Trace	デバッグモードでプログラムを1行ずつ実行します。関数が呼び出された場合は，その中にも入って1行ずつ実行していきます。
F8-Step	デバッグモードでプログラムを1行ずつ実行します。関数が呼び出された場合は，その中をトレースすることはしません。
F9-Make	Make（コンパイルとリンク）を行なって.EXE ファイルを作成します。
F10-Menu	アクティブウィンドウからメインメニューに移るか，あるいはメニューの中からアクティブウインドウに戻ります。

Watch ウィンドウ

Watch ウィンドウは，統合デバッガでプログラムを実行させる場合に，Message ウィンドウと置き換えられるものです。Watch ウィンドウには，監視式（プログラムから Watch ウィンドウに取り込んだ式）と各監視式のその時点での値が表示されます。監視式は各ステップごとに値が変化する可能性があるので，各ステップごとに評価されます。Watch ウィンドウを用いることにより，プログラムの実行中に，キーとなる重要な式の値を追跡していくことができます。

Watch ウィンドウに式を追加していくと，ウィンドウは TCINST の **R**esize windows

オプションで指定されている大きさに達するまでは拡大していきます。ウィンドウがその大きさに達しても式の追加はできますが，式をすべて見るには，***ROLLUP***，***ROLLDOWN***，上矢印，下矢印キーによってウィンドウをスクロールしなければならなくなります。

Watch ウィンドウ中の現在の式は，Watch ウィンドウがアクティブなときにはハイライト表示されたバーにより示され，アクティブでないときには左端にマークをつけて示されます。

Watch ウィンドウでの式の編集は，一般に Edit ウィンドウで使用する編集コマンドと同じものを用いて行なうことができます。たとえば，***CTRL-Y*** は監視式を1つ削除し，***CTRL-N*** は監視式を1つ挿入します。Watch ウィンドウでの基本的な編集コマンドを次の表に示します。

表 5.2 Watch ウィンドウの編集コマンド

キー	**機能**
CTRL-E または**上矢印**	カーソルを上へ移動します。
CTRL-X または**下矢印**	カーソルを下へ移動します。
CTRL-S または**左矢印**	監視式を左へスクロールします。
CTRL-D または**右矢印**	監視式を右へスクロールします。
リターン	監視式を編集します。
CTRL-N または ***INS***	監視式を挿入します。
CTRL-Y，***DEL***，***CTRL-G***	監視式を削除します。

カーソルが Watch ウィンドウにあるときには，クイックリファレンス行に表示されるホットキーの機能は次のようになります。

F1-Help ヘルプウィンドウを開きます。

F5-Zoom Watch ウィンドウを画面いっぱいに拡大します。

F6-Switch Edit ウィンドウをアクティブにします。

F10-Menu メインメニューに移ります。またメニューの中から Watch ウインドウに戻ります。

Ins Watch ウィンドウへ監視式を1つ追加します。

Del Watch ウィンドウから監視式を1つ削除します。

Ret Watch ウィンドウで現在指定されている監視式を編集します。

統合デバッガ

Turbo C 統合開発環境には，プログラム中のエラー（バグ）を見つけ出す手助けをする統合デバッガと呼ばれる特別なデバッガが組み込まれています。統合デバッガの使用方法の詳細については，このマニュアルの第4章を参照してください。ここでは，デバッグセッションを実行するために必要なメニューの機能を紹介します。

このデバッガでは，プログラムの実行を任意の場所で停止させることが可能であり，変数の値を見たり変更することができます。

デバッガのコントロール

デバッグするプログラムは，**O/C/C/OBJ** debug information と **Debug/Source** debugging トグルを On にしてコンパイルしたものでなければなりません。そうなっていれば，統合環境はプログラムを実行するときに自動的に統合デバッガを呼び出します。

デバッグセッションを **Run/Run** で開始すると，TC は（必要に応じて）ソースファイルのコンパイルおよびプログラムのリンクを行なって，プログラム実行のための準備を整えます。そして，ブレークポイントまたはプログラムの最後に達するまでプログラムを実行します。

ブレークポイントを設定していない状態でデバッグセッションを開始する場合は，***F8***（**Run/Step** over）を押してください。デバッガは関数 **main** の宣言のところで停止します。

TC がプログラムの実行準備を完了すれば，すでにデバッグセッションに入っており，TC デバッガの他のすべての機能が使用可能となります。

次のような形でプログラムを実行することができます。

- 1行ごとの実行（トレース）。関数呼び出しをスキップすることも，関数の中をトレースすることも可能です。
- 現在位置から，あらかじめ設定されているブレークポイントまでの実行。
- 現在位置から，カーソルを置いた場所までの実行。

これらの方法は，どんな組み合わせでも，どんな順序ででも使用することができます。

デバッグ中のプログラムのソースファイルを修正してから，そのまま実行を続けていくことは一般に賢明ではありません。ソースファイルを修正したら，**Compile/Make** EXE

file を選択してプログラムを再コンパイルしてください。実際にも，ソースファイルに変更を加えてから Step over や Trace into などの実行コマンドを選択すると，TC は.EXE ファイルを作成し直すかどうかを問い合わせてきます。作成し直されていれば，TC はソースファイルにさらに変更が加えられるまで何も問い合せてはきません。

デバッガ画面の表示

デバッガ画面では，上部に Edit ウィンドウ，下部に Watch ウィンドウが表示されます。***F6***を押すと，この2つのウィンドウの間を行き来することができます。

監視式が Watch ウィンドウに加えられていくと，Watch ウィンドウは（TCINST ユーティリティの Resize windows オプションで指定された）最大の大きさにまで拡大していき，最大の大きさに達するとスクロールしていきます。

プログラムの現在位置を**実行位置**と呼びます。この位置は，Edit ウィンドウの中で**実行バー**と呼ばれるハイライトバーにより示されています。

デバッグメニューコマンドとホットキー

表 5.3 にデバッガコマンドを示します。

表 5.3　デバッガコマンドとホットキー

ホットキー	メニューコマンド / 説明
F4	Run/Go to cursor プログラムを実行してカーソルのある行で停止します。デバッグセッションを開始します。
CTRL-F2	Run/Program reset 現在のデバッグセッションをキャンセルして，割り当てられたメモリを解放し，ファイルをクローズします。デバッグセッション内でのみ有効です。
F7	Run/Trace into 現在の関数の次の文を実行します。Debug/Source debugging と O/C/C/OBJ debug information トグルが On の状態でコンパイルされた関数

の呼び出しの場合には，その関数の中もトレースします。デバッグセッションを開始します。

F8 Run/Step over

現在の関数の次の文を実行します。呼び出される関数の内部はトレースしません。デバッグセッションを開始します。

O/C/C/Standard stack frame

Option/Compiler/Code generation/Standard stack frame トグルを切り換えます。Debug/Call stack オプションを正しく機能させるには，プログラムをコンパイルする際にこのトグルを On にしておかなければいけません。

O/C/C/OBJ debug information

O/C/C/OBJ debug information トグルを切り換えます。デバッグができるのはこのトグルを On に設定してコンパイル・リンクされたソースファイルだけです。

CTRL-F4 Debug/Evaluate

C の式を評価します。また変数の値を変更することもできます。

Debug/Find function

関数の定義を検索し，Edit ウィンドウに表示します。デバッグセッション内でのみ有効です。

CTRL-F3 Debug/Call stack

コールスタックを表示します。コールスタックから関数明を選択することで，その関数の現在実行中の行を表示させることができます。デバッグセッション内でのみ有効です。

Debug/Source debugging

デバッグを可能にするかどうかを制御します。これが On に設定されているときは統合デバッガとスタンドアロンのデバッガが両方とも使用できます。Standalone に設定されているときには，プログラムのデバッグにはスタンドアロンのデバッガしか使用できません。ただし TC でもプ

ログラムの実行は可能です。None に設定されている場合には.EXE ファイルにデバッグ情報が書き込まれないので，どちらのデバッガを使ってもプログラムのデバッグは行なえません。

CTRL-F7	Break/watch/**A**dd watch	監視式を1つ追加します。
	Break/watch/**D**elete watch	監視式を1つ削除します。
	Break/watch/**E**dit watch	監視式の編集を行ないます。
	Break/watch/**R**emove all watches	監視式をすべて削除します。
CTRL-F8	Break/watch/**T**oggle breakpoint カーソルのある行のブレークポイントを設定/解除します。	
	Break/watch/**C**lear breakpoint プログラム中のすべてのブレークポイントを解除します。	
	Break/watch/**V**iew next breakpoint 次のブレークポイントを表示します。	

表 5.4 にデバッガ実行中によく使用される他のメニューコマンドを示します。

表 5.4　デバッガとともに使われるメニューコマンドとホットキー

ホットキー	**メニューコマンド / 説明**
F5	アクティブウィンドウを全画面モードに拡大したり分割画面モードに戻したりします。
GRPH-F5	ディスプレイをユーザ画面に切り換えます。どれかキーを押すと統合環境に戻ります。
F6	Edit ウィンドウと，Watch ウィンドウまたは Message ウィンドウの間

でアクティブウィンドウを切り換えます。

GRPH-F6 Edit ウィンドウがアクティブな場合には，最後にエディタにロードされたファイルに切り換わります。下側のウィンドウがアクティブな場合は，Watch ウィンドウと Message ウィンドウの切り換えを行ないます。

CTRL-F9 **R**un/**R**un

デバッガ中で，あるいはデバッガなしでプログラムを実行します。必要な場合にはソースファイルのコンパイルやプログラムのリンクも行ないます。プログラムが **D**ebug/**S**ource debugging と **O**/**C**/**C**/**O**BJ debug information が On の状態でコンパイル・リンクされている場合には，ブレークポイントまであるいはプログラムの終わりまで実行します。

Project/**R**emove message

Message ウィンドウの内容を消去します。

第II部：メニューコマンド

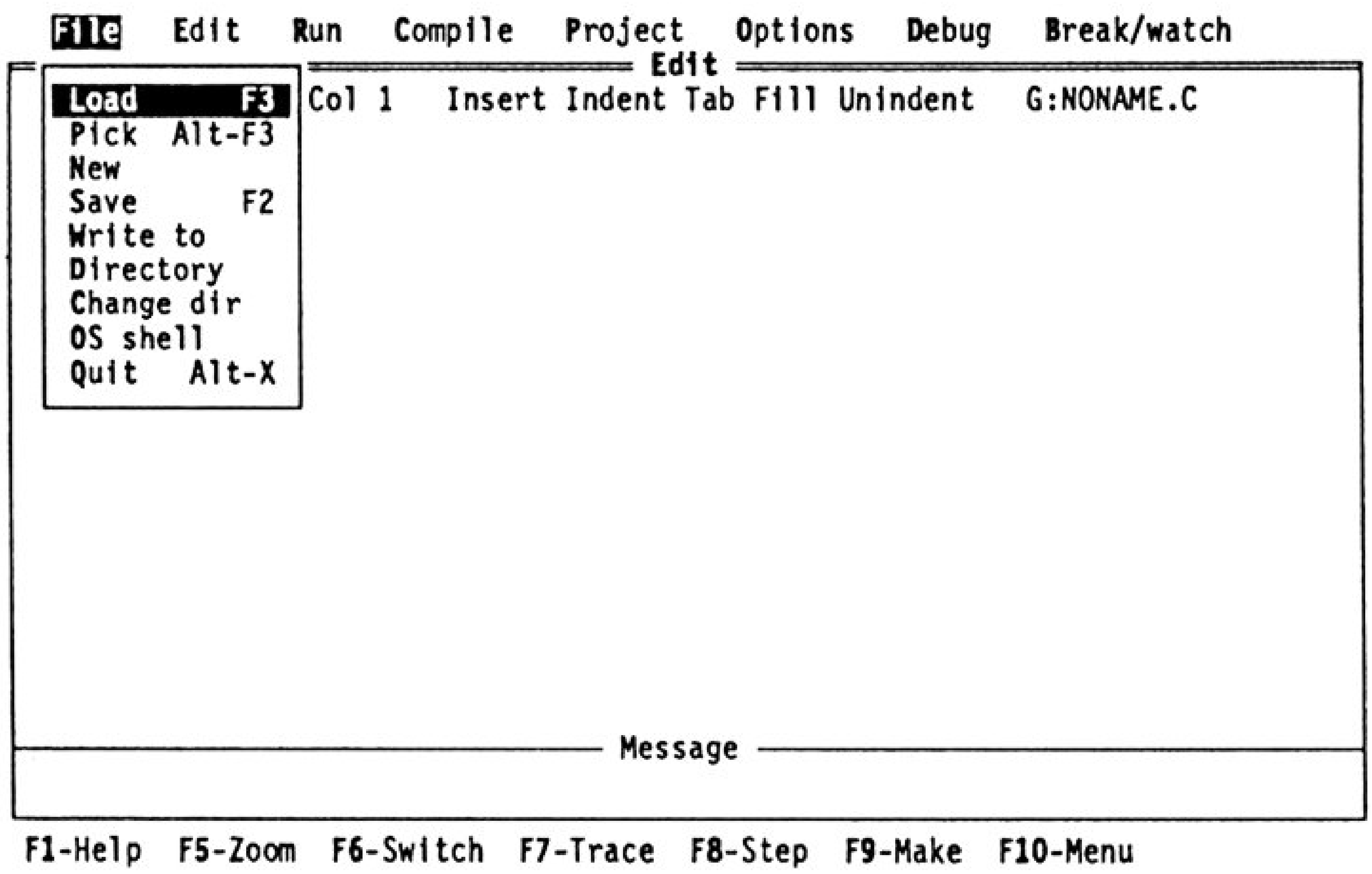
メインメニューには，ファイルのロードから，編集，コンパイル，デバッグ，プログラムの実行までに必要な機能が8つの項目に大別されて並んでいます。この8つの選択肢は，**File**，**Edit**，**Run**，**Compile**，**Project**，**Options**，**Debug**，**Break/watch** で，第II部ではこの8つの項目それぞれについて説明していきます。メインメニューの下に出てくるプルダウンメニューの中にあるオプションのいくつかは，上級のプログラミングで必要となるものなので，そうしたオプションについては第12章で詳しく説明することにします。

注意：この章で使っている“Make”はプロジェクト Make のことを指しており，スタンドアロンの MAKE ユーティリティのことではありません。プロジェクト Make は，MAKE に類似したプログラム作成ツールです。プロジェクト Make については第3章で，MAKE ユーティリティについては付録 C で詳しく解説しています。

File メニュー

File のプルダウンメニューの中には，既存のファイルのロードや，新しいファイルの作成，ファイルのセーブなどに関するいろいろなオプションがあります。あるファイルをロードすると，そのファイルはエディタに取り込まれます。ファイルに対する処理がすめば，そのファイルは好きなディレクトリに，好きなファイル名でセーブすることができます。さらに **File** メニューからは，カレントディレクトリの変更や，一時的に MS-DOS のシェルを呼び出して MS-DOS のコマンドを実行したりすることもできます。また，Turbo C を終了するオプションもこのメニューにあります。

図 5.4　File メニュー



Load（ロード――エディタに読み込む）

ファイルのロードを行ないます。ファイルの指定は，DOS 形式のワイルドカード（たとえば*.C）を使って選択するファイルの一覧表を表示させてからでもできますし，もちろん特定のファイル名を直接指定することもできます。どちらの場合でも，ロードしたいファイルの名前をそのままキー入力してください。

注意：間違ったドライブ名やディレクトリを入力すると，Error ボックスが表示されます。また，あるファイルを編集中で，そのファイルを変更したのにまだセーブしていない場合に，他のファイルをロードしようとすると，Verify（確認）ボックスが表示されます。いずれの場合でも，ボックスの中で指定されているキーを押すまでは，ホットキーは使えなくなります。

Pick（ピック――拾い上げる）

Edit ウィンドウにそれまでにロードされた8個までのファイルのピックリストから目的のファイルを選択します。選択されたファイルはエディタに取り込まれ，そのファイルを最後に編集したときの位置にカーソルが置かれます。リストの一番下の‘-- load file --’を選択すると，File/Load あるいは ***F3*** を選択したときと同様に Load File Name ボックスが

現われます。***GRPH-F3*** は，このピックリストを表示する短縮コマンドです。

ピックファイルを作っておけば，統合環境はこのファイル名のリストを次回の編集セッションまでセーブしておいてくれます。

ピックファイルの作り方については，Options/Directories/Pick file name の説明を参照してください。また，この章の第 III 部でピックファイルとピックリストについて詳しく解説しています。

New（ファイルの新規作成）

まったく新しいファイルを作成します。エディタ内はクリアされて，Edit ウィンドウに移ります。ファイル名は，デフォルトの NONAME.C になりますが，後でセーブするときに名前をつけ変えることができます。

Save（セーブ――格納）

エディタの中のファイルをディスクにセーブします。エディタ内のファイルの名前が NONAME.C の場合には Rename NONAME ボックスが表示され，名前を変更するかどうかを尋ねてきます。ホットキー ***F2*** を押せば，システムのどこからでもエディタ内のファイルをセーブすることができます。

Write to（ファイルへの書き出し）

ファイル名をたずねて，そのファイルにエディタの内容を書き出します。その名前のファイルがすでに存在しているときは，そのファイルに上書きしてよいかどうかを確認してきます。

Directory（ディレクトリの表示）

指定したディレクトリとファイルの一覧を表示します（カレントディレクトリの内容を見るのであれば単にリターンを押してください)。一覧が表示されてからでも，F4を押せばワイルドカードの指定を変更することができます。一覧の中からファイルを選択すると，そのファイルがエディタにロードされます。

Change dir（ディレクトリの変更）

カレントディレクトリが表示されて，ドライブとディレクトリが変更できるようになります。

OS shell（MS-DOS のシェル）

一時的に Turbo C から出て，MS-DOS のプロンプトに移行します。プロンプトから exit と入力すれば Turbo C に戻ることができます。Turbo C を終了せずに DOS コマンドを実行したいときに便利な機能です。

注意：デュアルモニタモードでは，DOS シェルはユーザ画面ではなく TC 画面に現われます。これによって，プログラムの出力をこわさずに DOS シェルを動かすことができます。

Quit（終了）

Turbo C を終了して，MS-DOS のコマンドレベルに戻ります。このコマンドのホットキーは ***GRPH-X*** です。

Edit コマンド

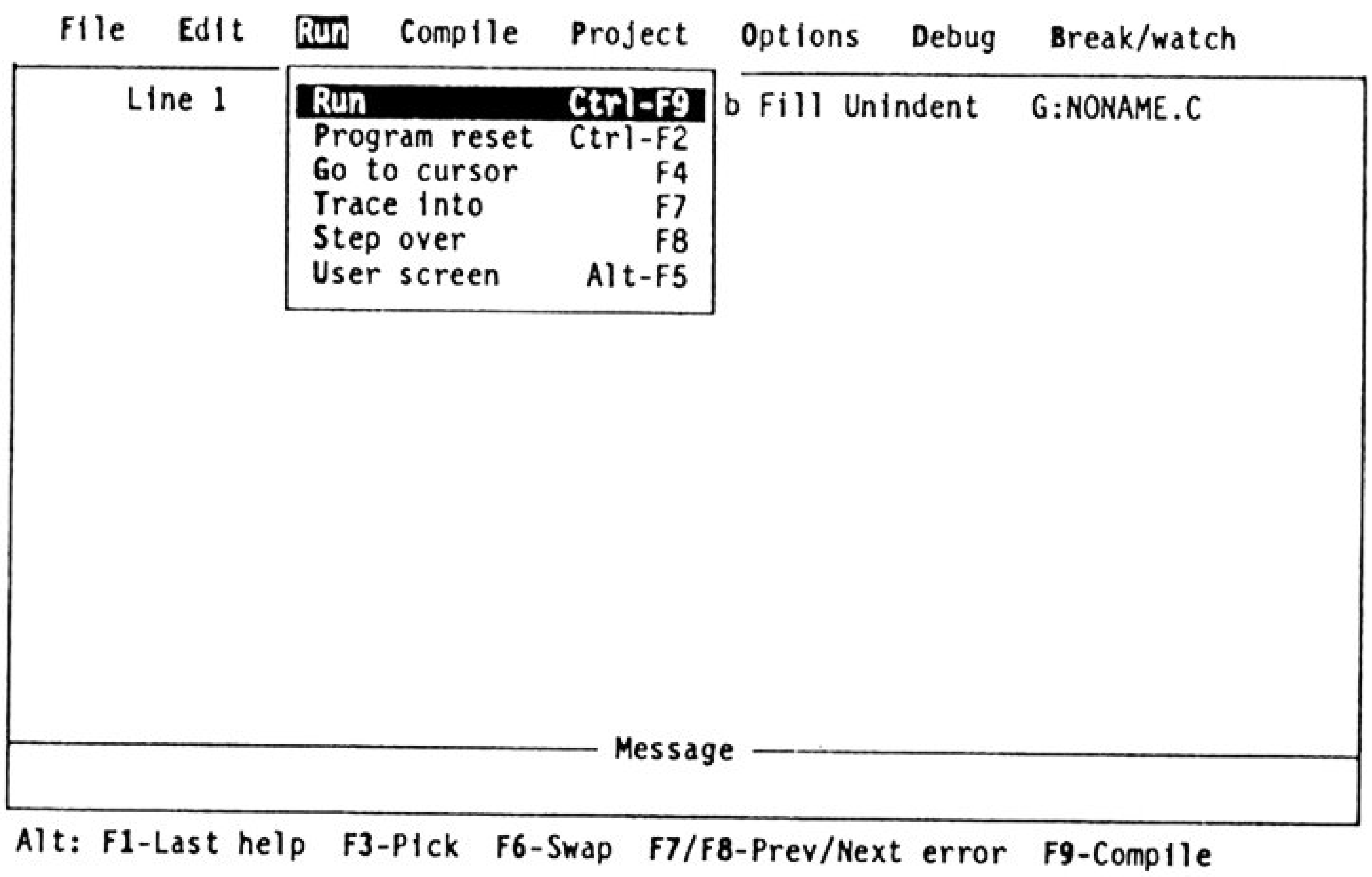
Edit コマンドは，組み込みのスクリーンエディタを呼び出します。

エディタの中からメインメニューを呼び出すには ***F10*** を押します（あるいは ***GRPH*** キーを押しながら，選択したいメインメニューのコマンドの頭文字を押します）。メインメニューに移ってもソースファイルは画面には表示されたままになっており，***ESC*** または ***E*** を押せばエディタに戻ることができます（システムのどこにいても，***GRPH-E*** を押せばエディタに移ることができます）。

Run メニュー

Run メニューのコマンドは，プログラムの実行や，デバッグセションの開始/終了を行ないます。Run/Run 以外のコマンドのいずれかを使用するには，Debug/Source debugging トグルを On に設定してプログラムをコンパイル・リンクしておかなければなりません。

図 5.5　Run メニュー



Run（実行）

Run/Run は，Options/Arguments によって渡された引数を用いて，プログラムを走らせます。ソースコードが，前回コンパイルされた後で修正されている場合は，Run/Run はプログラムをコンパイル・リンクし直すために，プロジェクト Make を呼び出します（プロジェクト Make は統合環境に組み込まれていプログラム作成ツールです。これについての詳細は第3章を参照してください）。

プログラムをデバッグしたくない場合は，Debug/Source debugging トグルを None か Standalone にセットしてコンパイル・リンクしてください。Source debugging トグルを On にしてプログラムをコンパイルすると，その結果できる実行可能コードにはデバッグ情報が書き込まれ，Run/Run コマンドの動作は次のように影響を受けます。

前回のコンパイル時からソースコードを修正していない場合：

■ **Run/Run** コマンドは，プログラムを次のブレークポイントまで，ブレークポイントが設定されていなければ最後まで走らせます。

前回のコンパイル時以降にソースコードを修正した場合：

■すでに **Run/Step over**（***F8***）か **Run/Trace into**（***F7***）によってプログラムのステップ実行を行なっている場合，**Run/Run** を選ぶと，プログラムを再作成（Build）するかどうかを問い合わせるプロンプトが画面に表示されます。

・Y を入力すると，プロジェクト Make がプログラムをコンパイル・リンクし直して，最初から実行します。

・N を入力すると，プログラムは次のブレークポイントまで，ブレークポイントが設定されていなければ最後まで実行します。

■まだプログラムのステップ実行を行なっていない場合，プロジェクト Make がプログラムを再コンパイルして最初から実行します。

Run/Run コマンドのホットキーは ***CTRL-F9*** です。

Program reset（プログラムのリセット）

Run/Program reset は現在のデバッグセションをキャンセルします。プログラムが割り当てたメモリを解放し，オープンされているすべてのファイルをクローズします。

Run/Program reset のホットキーは ***CTRL-F2*** です。

Go to cursor（カーソル位置まで実行）

Run/Go to cursor を選択すると，現在実行バーがあるところから Edit ウィンドウのカーソルがある行まで，プログラムを走らせます。カーソルが実行可能な文を含まない行の上にあれば，コマンドは Esc ボックスを出して警告を発します。**Run/Go to cursor** はデバッグセションを開始します。

Go to cursor は，永久的なブレークポイントは設定しませんが，カーソル位置より前に永久的なブレークポイントを見つけると，そこでプログラムを止めます。この場合は，もう一度 Go to cursor コマンドを選択する必要があります。

プログラムのデバッグしたい部分に実行バーを進めるには，Go to cursor を使用してください。プログラムが特定の文に来るたびに止めたい場合は，その行にブレークポイントを設定してください（**Break/watch/Toggle breakpoint** またはホットキー ***CTRL-F8***）。

Run/Go to cursor のホットキーは ***F4*** です。

Trace into（トレース）

Run/Trace into は，現在の関数における次の文を実行します。その文に，デバッガがアクセスできる関数の呼び出しが含まれていなければ，**Trace into** は次の実行可能な文のところで止まります。

その文に，デバッガがアクセスできる関数の呼び出しが含まれていれば，**Trace into** はその関数の定義の最初のところで止まります。引き続いて **Trace into** あるいは **Step over** コマンドを入力すれば，その関数の中の文が実行されます。デバッガはその関数から出たときに，その関数を呼び出した文の評価を再開します。

デバッガがアクセスできる関数は，**O/C/C/OBJ debug information** トグルと **Debug/Source debugging** トグルをともに On に設定してコンパイルされたソースファイル内で定義されていて，かつそのソースファイルがディスク上に存在しているものだけです。

現在デバッグ中の関数内で呼び出されている関数の中に実行位置を移したい場合は，**Trace into** を使用してください。

Run/Trace into のホットキーは ***F7*** です。

Step over（ステップ実行）

Run/Step over は，現在の関数内の次の文を実行します。他の関数を呼び出す文があっても，その中はトレースしません。たとえ，デバッガがそれらの関数をアクセス可能であってもです。

現在デバッグしている関数を1度に1文ずつ走らせたいときは，**Step over** を使用してください。

Run/Trace into と **Run/Step over** の違いを示す例を示します。エディタにロードされているプログラムの初めの12行が次のようになっているとします。

```
int findit(void)                          /*  1行 */
{
    return(2);                            /*  3行 */
}

void main()                               /*  6行 */
{
    int i, j;                             /*  8行 */

    i = fintit();                         /* 10行 */
    printf("%d¥n", i);                    /* 11行 */
    j = 0;   . . .                        /* 12行 */
```

findit は，デバッグ情報付きでコンパイルされたモジュール内のユーザ定義関数です。いま実行バーがプログラムの10行目にあるとしましょう。

- **Run/Trace into** を選ぶと，実行バーは *findit* 関数の最初の行（プログラムの1行目）に移動し，関数のステップ実行を行なうことができます。
- **Run/Step over** を選ぶと，*findit* 関数が実行されて戻り値が *i* に代入されます。そして，実行バーは11行目に移動します。

実行バーがプログラムの11行目にあった場合は，どちらのコマンドを選んでも違いはありません。**Run/Trace into** も **Run/Step over** も **printf** 関数を実行し，実行バーを12行目に移動させるはずです。これは，**printf** 関数がデバッグ情報を含んでいないためです。

Run/Step over のホットキーは ***F8*** です。

User screen（ユーザ画面）

統合環境の画面からユーザ画面に切り換えます。

Turbo C 統合環境下で実行されるプログラムが行なう画面出力は，すべてユーザ画面に送られます。**Run/Run** コマンドによってプログラムを実行すると，統合環境の画面からユーザ画面に切り換わってからプログラムが実行が始まりますが，そのプログラムがキーボードからの入力を行なわない場合，プログラムの終了とともに統合環境の画面に即座に戻ってしまいます(従来のバージョンのように一時停止はしません)。ただし，プログラムが出力を行なったユーザ画面は常に保存されており，このコマンドを呼び出すことによっていつでも表示させることができます。どれかキーを押せば統合環境の画面に戻ります。

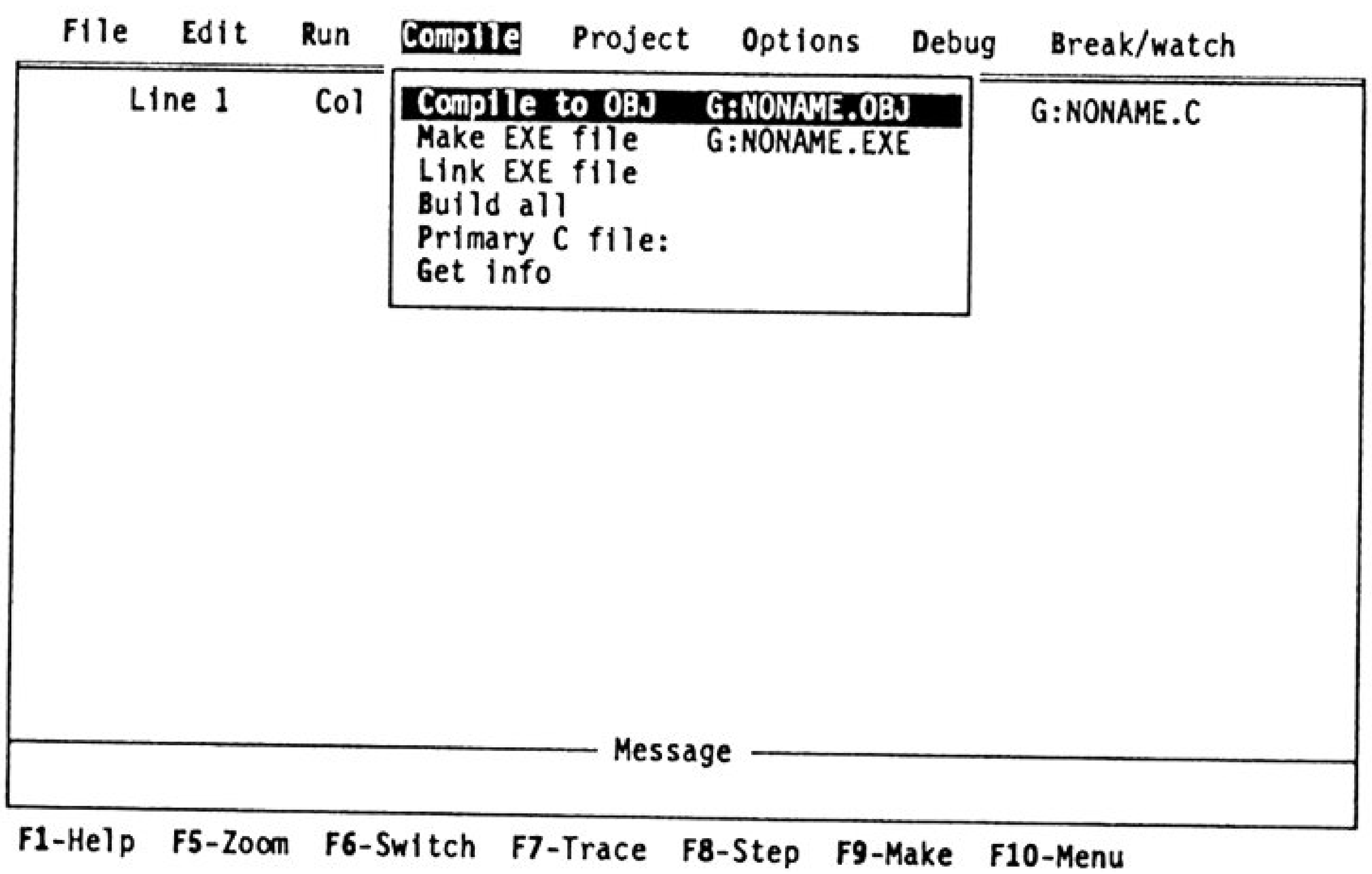
Run/User screen のホットキーは ***GRPH-F5*** です。

Compile メニュー

Compile メニューには以下の6つの項目があります。

- ■ Compile to OBJ　.OBJ ファイルを作成する
- ■ Make EXE file　Make して. EXE ファイルを作成する
- ■ Link EXE file　リンクして. EXE ファイルを作成する
- ■ Build all　プロジェクトのすべてのファイルを作成（Build）する
- ■ Primary C file　プライマリファイルを設定する
- ■ Get info　前回のコンパイルに関する情報を表示する

図 5.6　Compile メニュー



Compile to OBJ（.OBJ ファイルを作成する）

このコマンドは，.C ファイルを.OBJ ファイルにコンパイルします。必らず右側に作成されるファイルの名前（A:EXAMPLE.OBJ など）が表示されています。ここに表示される.OBJ ファイルの名前は次の順序で決められます。

1. Primary C file で指定されていればその名前。
2. 1の指定がなければ，Edit ウィンドウ内のファイルの名前。

Turbo C がコンパイルを行なっているときには，ウィンドウがポップアップしてコンパイルの経過および結果を表示します。コンパイル/Make が完了したら，どれかキーを押してこのウィンドウを消去してください。エラーが起きた場合は，自動的に Message ウィンドウに入って最初のエラー（ハイライト表示されます）に位置づけられます。

このコマンドのホットキーは ***GRPH-F9*** です。

Make EXE file（Make して EXE ファイルを作成する）

このコマンドは，プロジェクト Make を呼び出して.EXE ファイルを Make します。右側には必らず，作成される.EXE ファイルの名前（A:EXAMPLE.EXE など）が表示されています。

.EXE ファイルの名前は次の順序で決められます。

1. Project/Project name で指定されているプロジェクトファイル名（.PRJ)。
2. 1の指定がなければ，Primary C file で指定されているファイル名。
3. 2の指定がなければ，Edit ウィンドウに最後にロードされたファイルの名前。

このコマンドのホットキーは ***F9*** です。

Link EXE file（リンクして EXE ファイルを作成する）

現在の.OBJ ファイルと，.LIB ファイル(デフォルト，あるいは現在のプロジェクトファイルで定義されているもの）をリンクして，新しい.EXE ファイルを作成します。

Build all（プロジェクトのすべてを Build する）

プロジェクトに含まれるファイルを，作成日時に関係なくすべて再作成（Build）します。このコマンドは，無条件にでないこと以外は Make EXE file とよく似ています。Make EXE file は最新でないものだけを再作成します。このコマンドは最初に，そのプロジェクトのすべての.OBJ ファイルの日時をゼロにセットし，それから Make を行ないます(したがって，Build all コマンドを ***CTRL-STOP*** で中止させた場合，Compile/Make EXE file を選択すれば，中止した地点から再開させることになります。

Primary C file（プライマリファイルを設定する）

このオプションでは，Compile/Compile to OBJ コマンド（***GRPH-F9***）を選択したときに，.OBJ ファイルにコンパイルされるの.C ファイル（プライマリファイル）を指定することができます。

Primary C file コマンドは，1つの.C ファイルと複数のインクルード（ヘッダ）ファイル（.H）で構成されるプログラムをコンパイルする場合に便利です(なければ困るというものではありませんが)。コンパイル中にエラーが発生すると，エラーを含むファイル（.C ファイルか.H ファイル）が自動的にエディタにロードされて，すぐに修正が行なえるようになります。そのあと ***GRPH-F9*** を押せば，エディタ内にはなくてもプライマリファイルが再コンパイルされます。

注意：.H ファイルは，Options/Environment/Message tracking が All files になっている場合にのみ自動的にロードされます。デフォルトの設定（Current file）の場合には自動的なロードは行なわれません。

Get info（情報を得る）

Compile/Get info は，以下の情報を表示するウィンドウを呼び出します。

■プライマリファイル

■現在のファイルに対応するオブジェクトファイル名

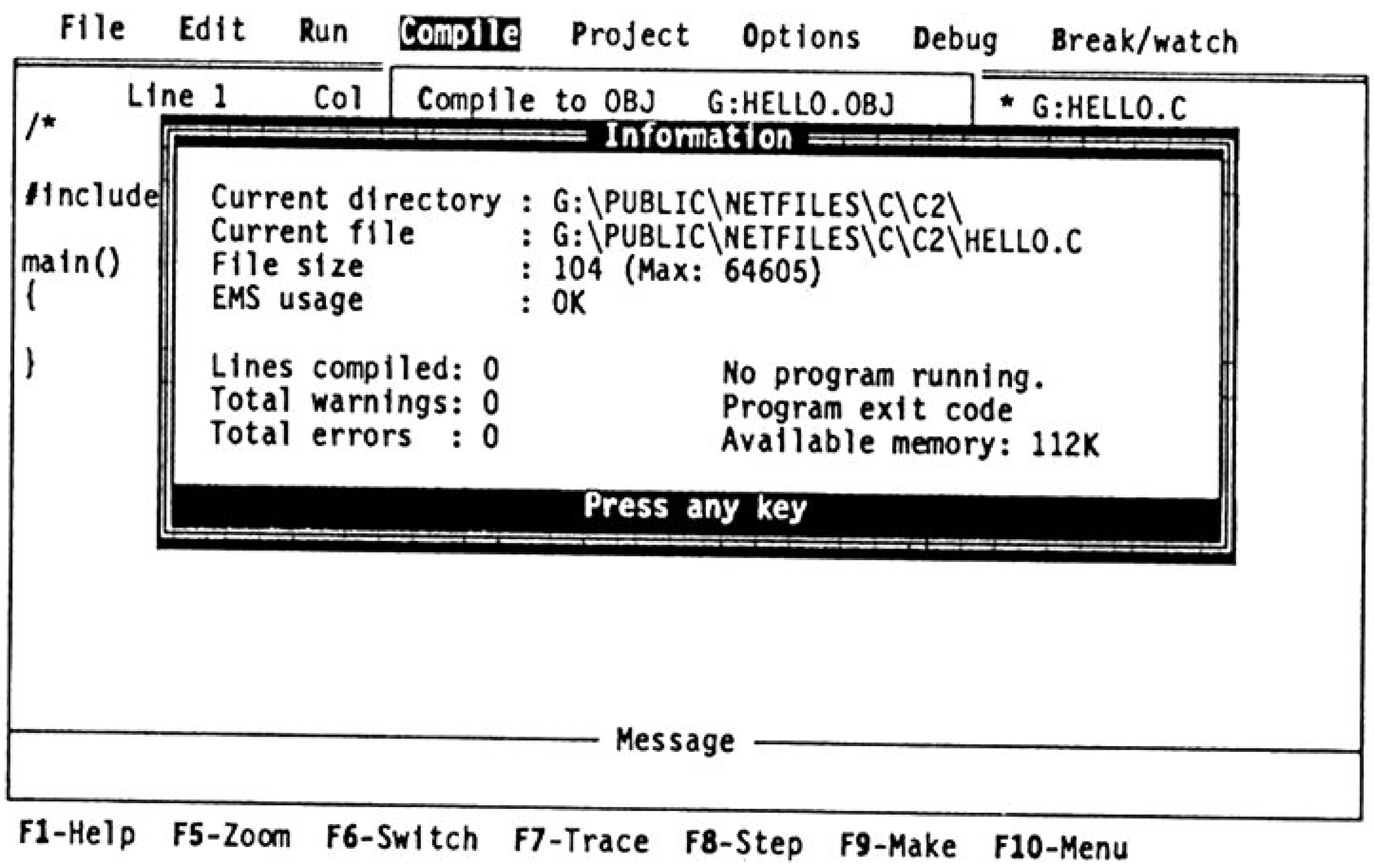
■現在のソースファイル名

■現在のソースファイルのサイズ（バイト数）

■プログラムの終了コード

■使用可能なメモリ

図 5.7　Comiple/Get info の画面

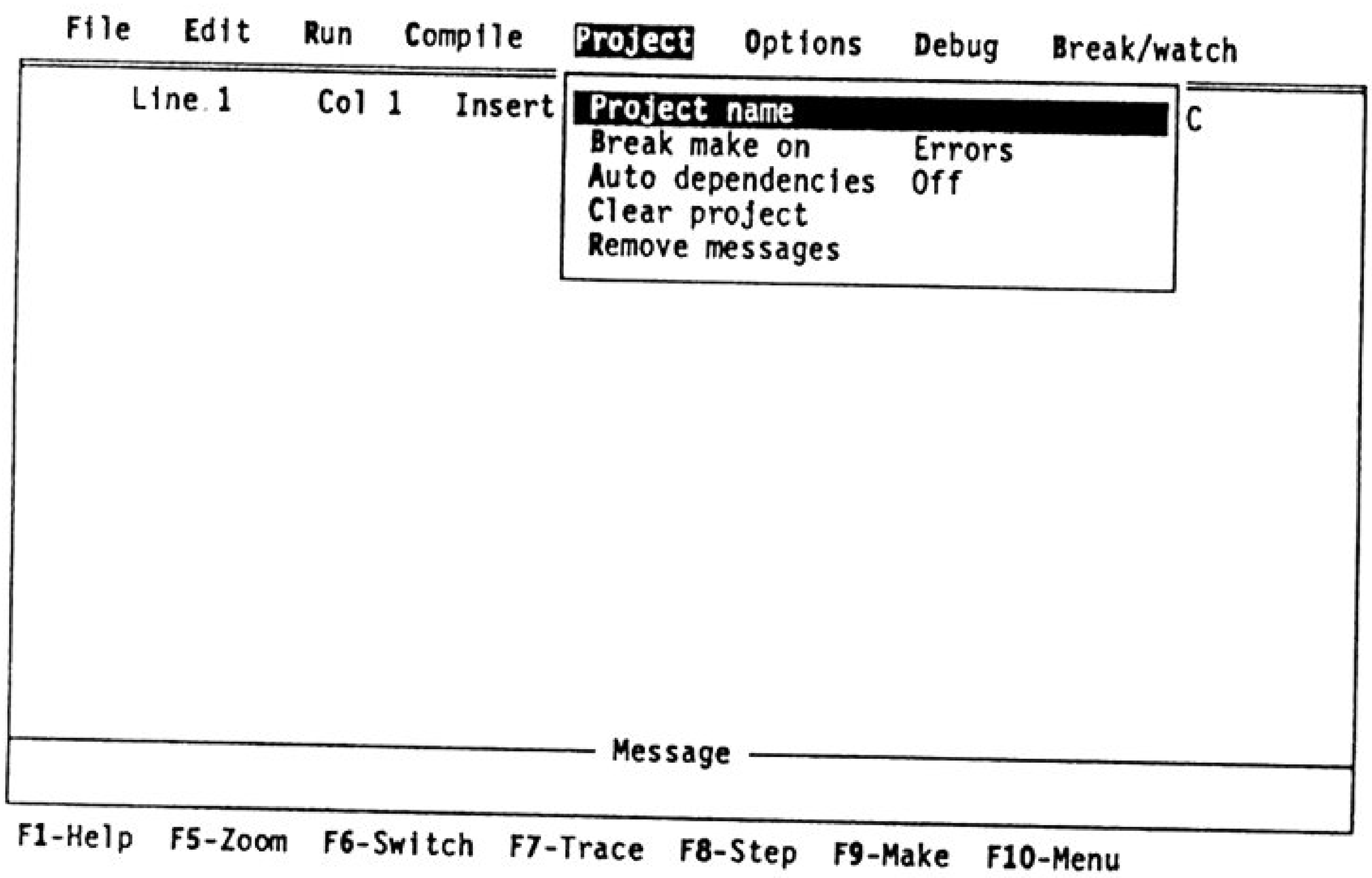


Project メニュー

Project メニューのコマンドでは，複数のソースファイルやオブジェクトファイルをまとめて最終的なプログラムを作成する際の設定を行ないます。

プロジェクトに関しては第3章で詳しく説明しています。

図 5.8　Project メニュー



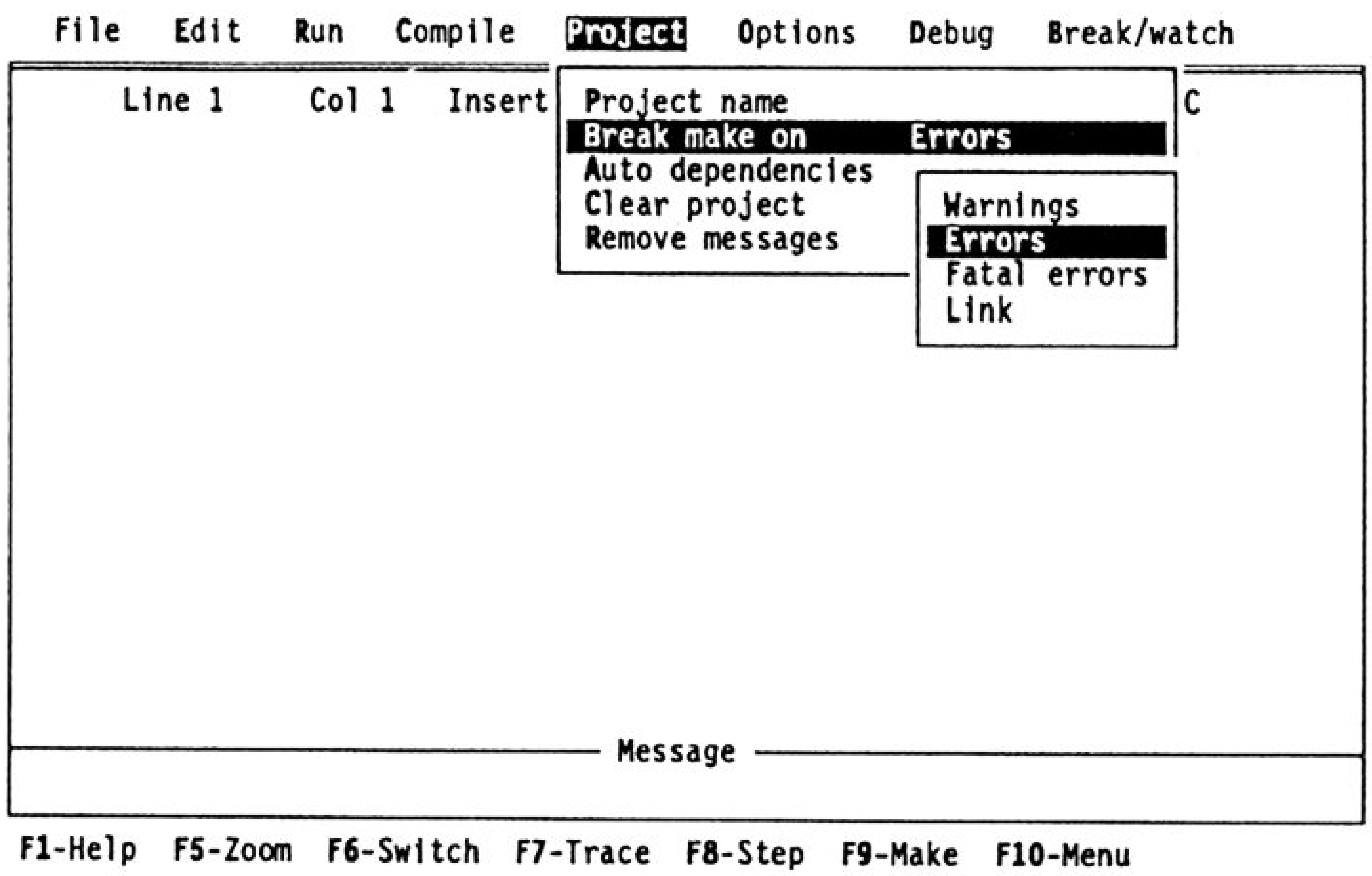
Project name（プロジェクト名）

プロジェクトファイル（コンパイルおよびリンクするファイルの名前が書き込まれている）の名前を選択します。プロジェクト名は，作成される.EXE ファイルや.MAP ファイルの名前としても使用されます。通常プロジェクトファイルには拡張子.PRJ をつけます。

Break make on（Make の停止レベル）

このメニューでは，コンパイラの処理をどの段階で停止させるかを，Warnings（警告），Errors（エラー），Fatal errors（致命的なエラー），Link（リンク）の中から指定します。

図 5.9　Project/Break make on メニュー



Auto dependencies（自動依存チェック）

このオプションはトグルです。On にセットすると，プロジェクト Make はディスク上の各.OBJ ファイルとプロジェクトファイル内の対応するソースファイルの依存関係を自動的にチェックします。

プロジェクト Make は.OBJ ファイルをオープンして，その.OBJ ファイルのソースコードが含まれているファイルに関する情報を探します。この情報は，TC および TCC でソースモジュールがコンパイルされるときには，必らず.OBJ ファイルの中に書き込まれます。そして，.OBJ ファイルを作成するために使用された各ファイルの日付・時刻が，.OBJ ファイル内の日付・時刻の情報と比較されます。その日付・時刻が異なっていれば，ソースファイルが再コンパイルされます。

この機能は，自動依存チェックと呼ばれます。

Auto dependencies オプションが Off の場合には，このようなファイルチェックは行なわれません。

Clear project（プロジェクトのクリア）

このコマンドは，プロジェクト名をクリアし，Message ウィンドウをリセットします。

Remove Messages（メッセージの消去）

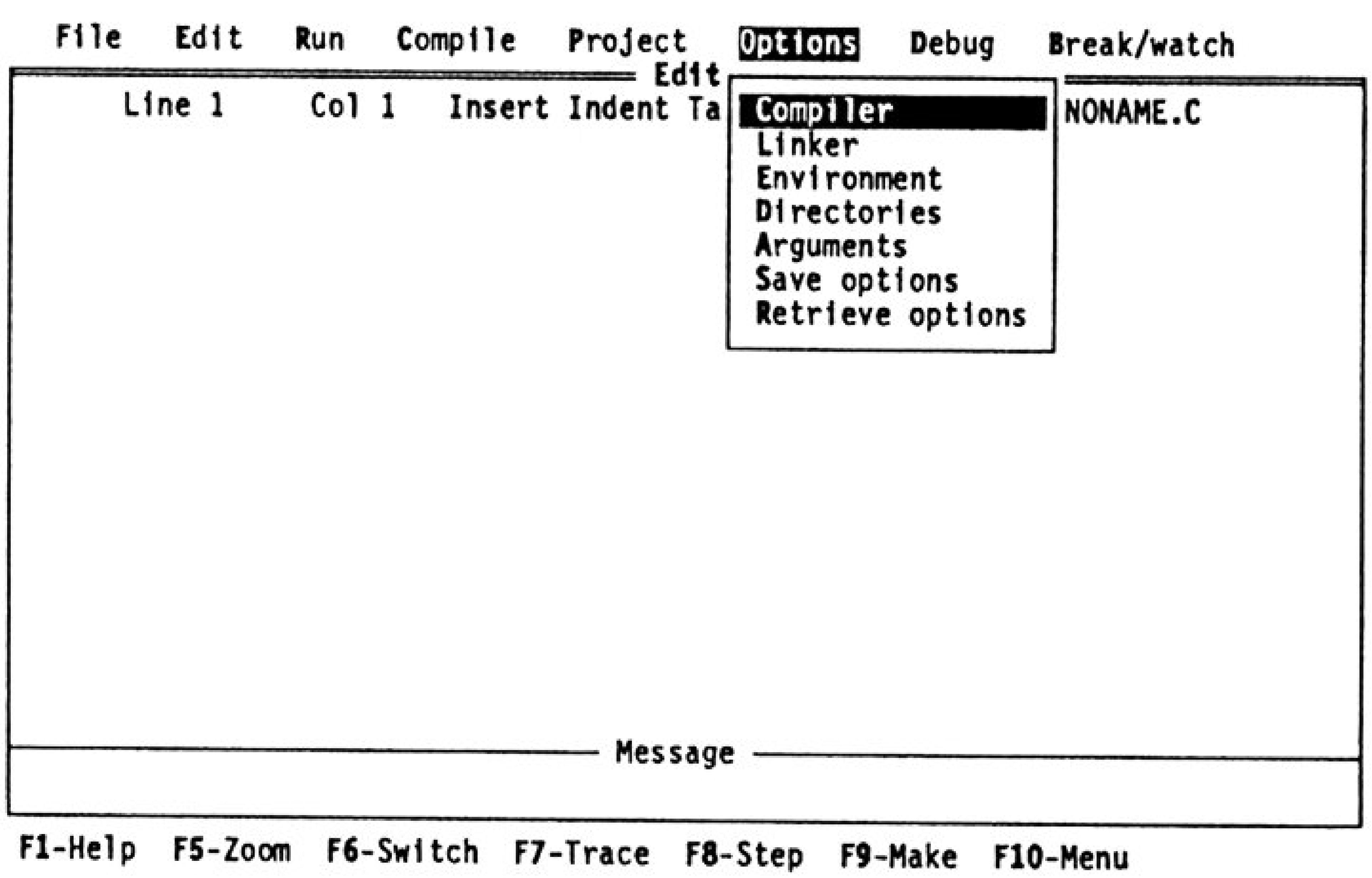
このコマンドは，Message ウィンドウからすべてのメッセージをクリアします。

Options メニュー

Options メニューでは，統合環境の働きに関する設定を行ないます。コンパイラやリンカのオプション，ライブラリやインクルードのディレクトリ，プログラムの実行時引数などを指定することができます。このメニューの項目には，以下に示すように，さらにメニューを呼び出すものが4つ，設定項目が1つ，管理面の仕事を行なうコマンドが2つあります。

- ■ Compiler（さらにメニューあり）
- ■ Linker（さらにメニューあり）
- ■ Environment（さらにメニューあり）
- ■ Directories（さらにメニューあり）
- ■ Arguments（設定）
- ■ Save options（仕事を行なう）
- ■ Retrieve options（仕事を行なう）

図 5.10　Options メニュー

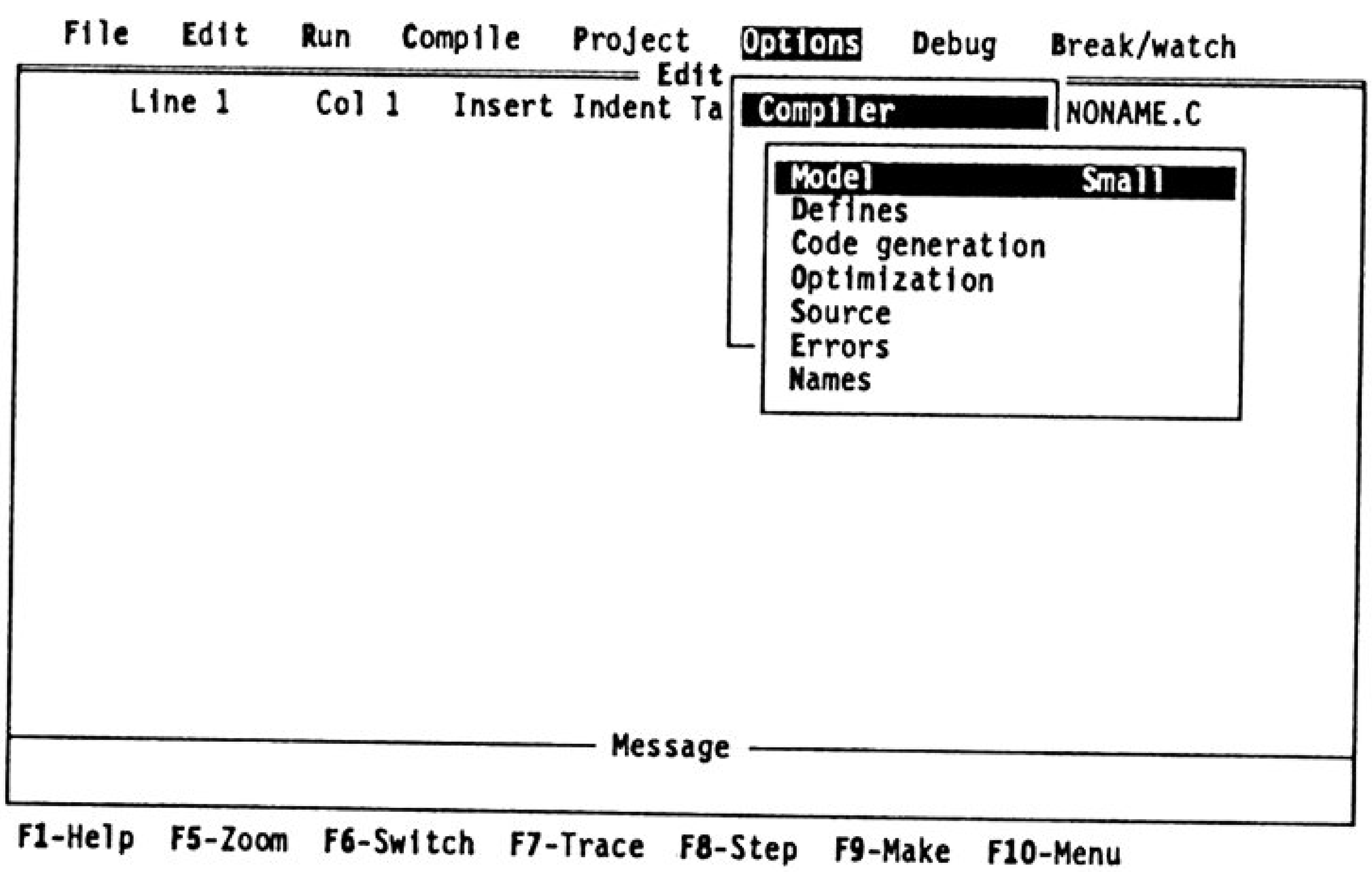


Compiler（コンパイラ）

このメニューの中のオプションでは，ハードウェアに関連するコンフィギュレーション，メモリモデルの指定，デバッグの手法やコードの最適化に関する指定，診断メッセージの制御，およびマクロの定義を行ないます。この後の何ページかで説明するメニュー項目は次のとおりです。

- ■ Model（メモリモデル）
- ■ Defines（マクロ定義）
- ■ Code generation（コード生成）
- ■ Optimization（最適化）
- ■ Source（ソースファイル）
- ■ Errors（エラー）
- ■ Names（セクションの名前）

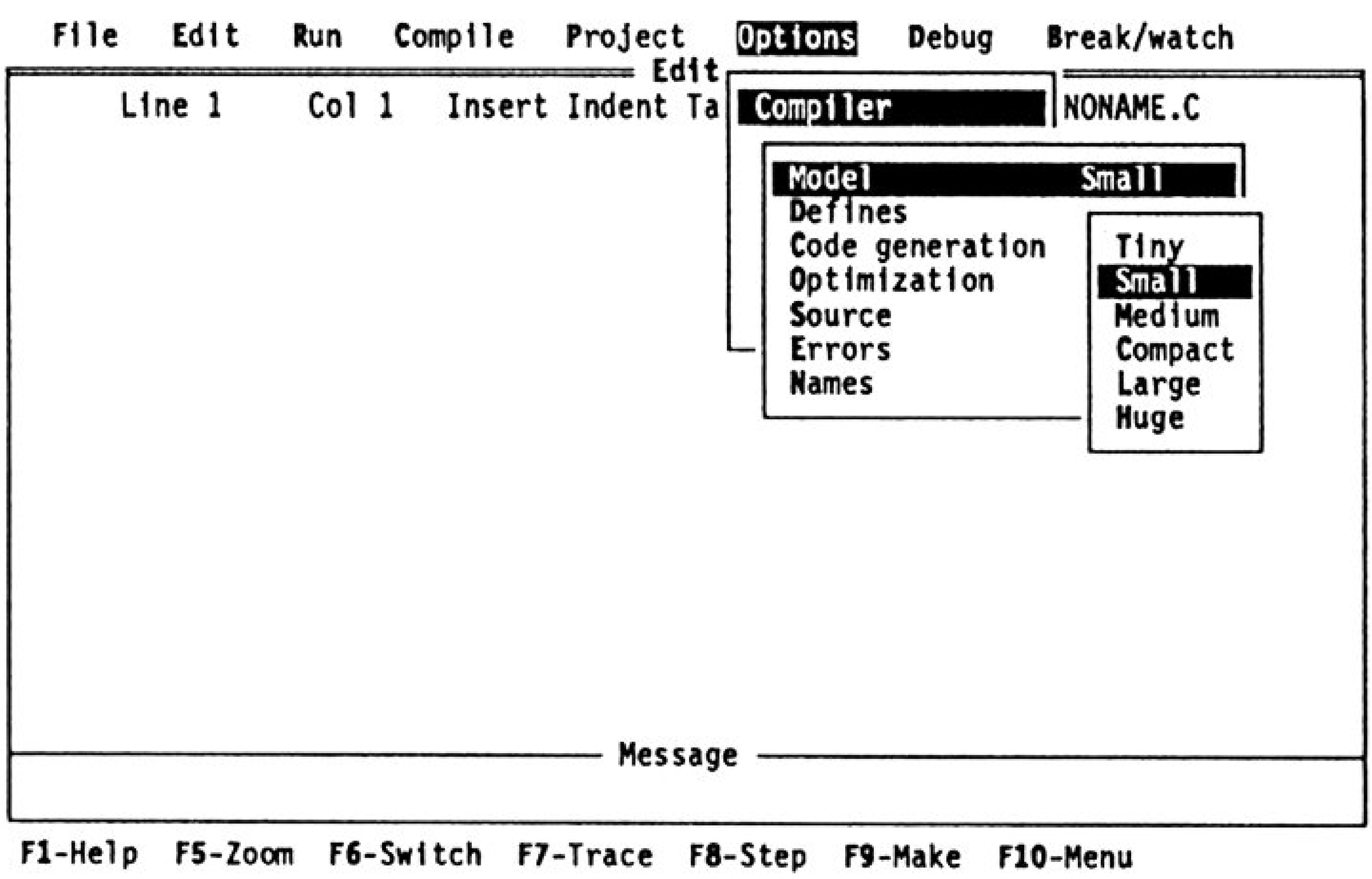
図 5.11　Options/Compiler メニュー



Model メニュー（メモリモデル）

このメニューは，Turbo C で使用できる6種類のメモリモデルを選択するスイッチになっています。指定されたメモリモデルにしたがって，デフォルトのメモリアドレッシング方式が決められます。指定できるオプションは，Tiny，Small，Compact，Medium，Large，Huge です。デフォルトのメモリモデルは Small なので，Model の右側には，選択されているメモリモデルとして Small と表示されています。各メモリモデルの詳細については，第12章で詳しく説明しています。

図 5.12　Options/Compiler/Model メニュー



Defines（マクロ定義）

Defines を選択すると，マクロ定義のボックスが現われ，プリプロセッサに渡すマクロを定義することができます。セミコロン（;）で区切れば，複数の定義を行なうことができます。また，等号（=）を使ってシンボルに値を代入することもできます。

先頭と末尾の空白は取り除かれますが，途中の空白はそのままになります。マクロの中にセミコロンを含める場合は，その前に円記号（¥）をつけてください（¥; とします）。

下に示したのはマクロ定義の例です。

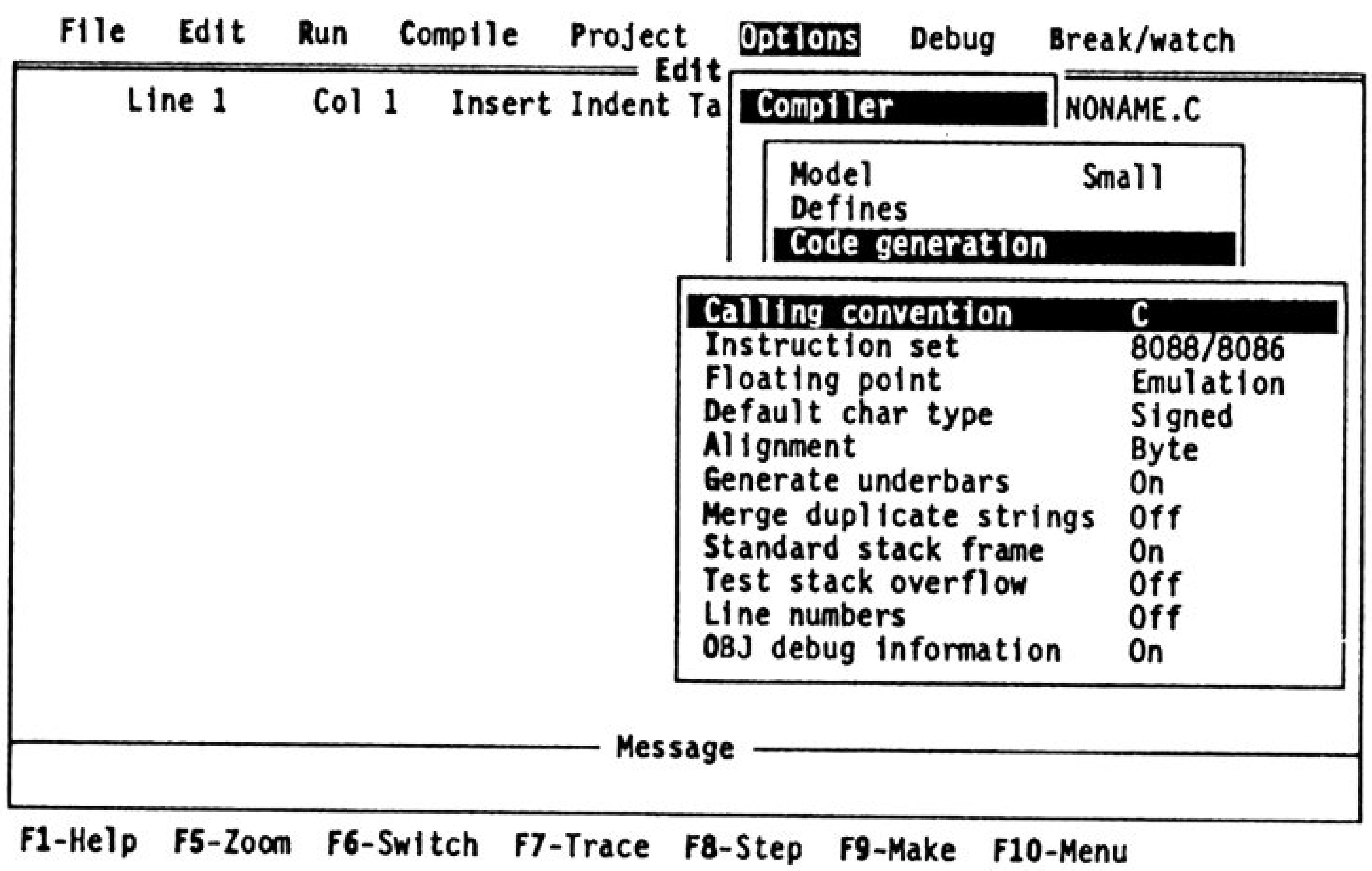
```
BETA_TEST; ONE = 1; COMPILER = TURBOC
```

ここでは，シンボル *BETA _TEST* を定義し，*ONE* に *1*, *COMPILER* に文字列 *TURBOC* を代入しています。

Code generation メニュー（コード生成）

このメニューから，コンパイラがどのようなオブジェクトコードを生成するのかを指示します。

図 5.13　Options/Compiler/Code generation メニュー



Calling convension（関数の呼び出し方法）：

関数を呼び出すときに，C の呼び出し方法(コーリングシーケンス)を使うか，Pascal の（速い）方法で行なうかをコンパイラに指示します。C と Pascal の関数呼び出しの違いは，スタックのクリアのしかた，受け渡される引数の個数と順序，大文字/小文字の扱い，および外部名のプレフィクス（下線）などに関するものです。

注意：このオプションは，第12章を読み終えてよく理解された方以外は変更しないようにしてください。

Instruction set（**命令セット**）：

対象となるCPUの種類を指定します。8088/8086と80186/80286のどちらの命令セットかを選ぶトグルになっています。デフォルトでは8088/8086のコードを生成します。PC-9801VMなどCPUがV30の場合には8088/8086を選んでください。Turbo Cは80186/286の拡張命令を生成することができます。またこのオプションによって，PC-9801VX上のMS-DOS 3.xの非保護モードで走る80286のプログラムを生成することもできます。

Floating point（**浮動小数点**）：

以下の3つのオプションがあります。

- ■8087/80287　8087のインラインコードを直接生成します。
- ■ Emulation　8087/80287があるかどうかチェックして，もしあればそれを使います。なければ8087をエミュレートします（精度は変わりませんがスピードは落ちます）。
- ■ None　浮動小数点をいっさい使用しないことを指定します（Noneとしているにもかかわらず，プログラム内で浮動小数点演算を行なっていると，リンクエラーが起こります）。

Default char type（**char型のデフォルト**）：

char宣言子が，符号つき（Signed）と符号なし（Unsigned）のどちらを表わすかを指定するトグルです。Signedとすると，**char**宣言子はすべて**signed char**として扱われ，Unsignedにすると**unsigned char**として扱われます。デフォルトは**signed**です。

Alignment（**アラインメント－メモリ割り当ての境界**）：

これは，メモリ割り当てをワード境界（Word）で行なうか，バイト境界（Byte）で行なうかを指定するトグルになっています。ワード境界の場合は，文字データ以外はすべて偶数アドレスに配置されます。バイト境界の場合には，データは偶数アドレス/奇数アドレスの区別なく配置されます。ワード境界にしておくと，8086や80286がメモリ上のデータを読んだり，メモリにデータを書き込むスピードが速くなります。

Generate underbars（**シンボルの下線を生成する**）：

これはトグルスイッチで，デフォルトはOnになっています。

注意：このオプションは，第12章を読み終えてよく理解された方以外は変更しないようにしてください。

Merge duplicate strings（**同一文字列の併合**）：

これは最適化の一種で，ある文字列が他の文字列と一致している場合に，それを併合します。このオプションをOnにしておくと，作成されるプログラムが小さくなります。デフォルトはOffです。

Standard stack frame（**標準のスタックフレーム**）：

Onのときには，標準のスタックフレーム（関数の標準エントリおよび標準終了コード）を生成します。デバッガを使用するときにはなかなか便利で，呼び出されたサブルーチンのスタックを通してトレースバックする過程を簡略化することができます。デフォルトはOffです。

Standard stack frameオプションはトグルです。このオプションをOffにしてソースファイルをコンパイルすると，ローカル変数を使用せず，かつパラメータをとらない関数のエントリおよび終了コードは，簡略化されたコードにコンパイルされます。これによってコードは小さくまた速くなりますが，**Debug/Call stack**によってその関数を"見る"ことはできなくなります。したがって，デバッグのためにコンパイルするときには，このトグルは常にOnにセットしておくべきです。

Test stack overflow（**スタックオーバーフローのテスト**）：

実行時にスタックオーバーフローのチェックを行なうコードを生成します。これによってプログラムのサイズと速度は犠牲になりますが，スタックオーバーフローは発見の難しいバグであり，開発過程ではたいへん有効なオプションです。デフォルトはOffです。

Line numbers（**行番号**）：

オブジェクトおよびマップファイルの中に行番号を含めるかどうかを指定するトグルスイッチです（この情報はシンボリックデバッガが使用します）。行番号を含めると，.OBJおよび.MAPファイルのサイズは大きくなりますが，実行可能プログラムのサイズと速度には影響を与えません（**D**ebug/**S**ource debuggingトグルをOnにセットし，**O**/**C**/**C**/**L**ine numbersトグルがOnの状態で生成されたオブジェクトファイルをリンクすれば，実行可能ファイルのサイズは大きくなります。増加したサイズはデバッグ情報によるものです）。

Options/**C**ompiler/**O**ptimization/**J**ump optimizationがOnになっていると，ソースプログラムの何行かがグループとしてまとめられたり，行が並べ替えられたりすることがあるので，このオプションを使うときには**J**ump optimizationはOffにしておいた方がよいでしょう（行番号を追うのが困難になります）。

OBJ debug information（OBJ ファイルのデバッグ情報）：

デバッグ情報を.OBJ ファイルに含めるかどうかを制御します。このトグルはデフォルトでは On で，TC 統合デバッガおよびスタンドアロンの Turbo デバッガによるデバッグが可能になります。

Kanji strings（漢字の文字列）：

文字列リテラルを処理するときに，その中に含まれる0x80～0x9F，および0xE0～0xFC の範囲の8ビットコードをどのように扱うかを指定するトグルスイッチです。デフォルトでは On になっており，この範囲のコードを全角文字（2バイトコード）の1バイト目として扱います。また，その次の1バイト（全角文字の2バイト目）が円記号（¥，0x5C）の場合に，これをエスケープシーケンスの開始として扱わないようになります。

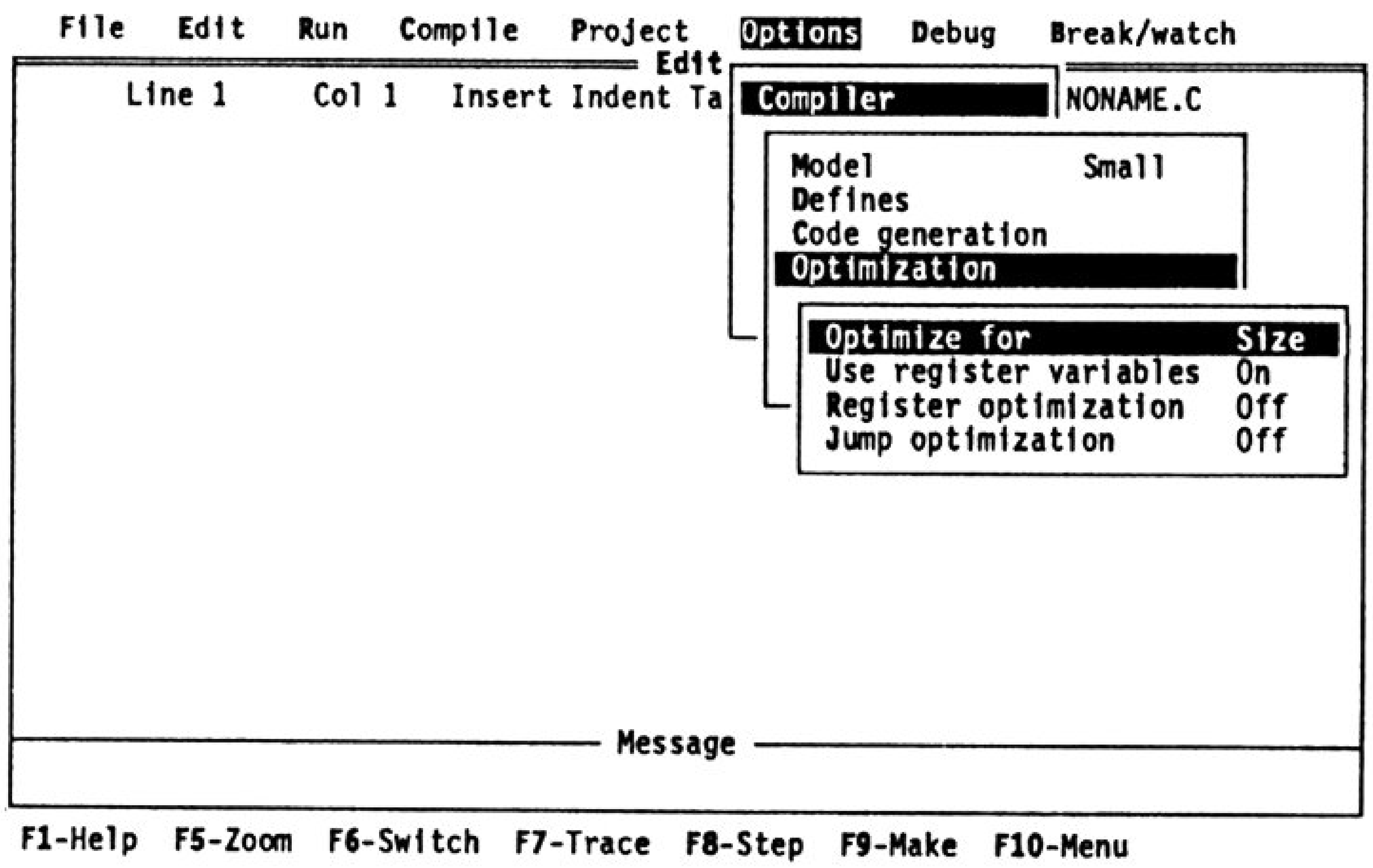
Off にすると，0x80～0x9F および0xE0～0xFC の範囲のコードはグラフィック文字として扱われ，すべてのコードは1バイト単位で処理されます。

デフォルトは On です。

Optimization メニュー（最適化）

このメニューでは，プログラミングの必要に応じて最適化の方法を指定することができます。

図 5.14　Options/Compiler/Optimization メニュー



Optimize for（最適化の方針）：

Turbo C がコードを生成する方針を指定します。通常は Size（プログラムの大きさ）が指定されており，最も小さなコードが生成されるようになっています。このトグルを Speed（速度）に設定すると，要求される仕事を最も速く行なえるコードが生成されます。

Use register variables（レジスタ変数の使用）：

レジスタ変数を使用するかどうかを指定します。このオプションが On になっていると，自動的にレジスタ変数が割り当てられます。Off の場合には，プログラム内で register 宣言が行なわれていてもレジスタ変数は使用されません。詳しくは付録 B を参照してください。

一般的には，レジスタ変数をサポートしない既存のアセンブラルーチンとのインターフェースをとるのでなければ，このオプションは On にしておくのがよいでしょう。

Register optimization（レジスタ使用の最適化）:

レジスタの内容を記憶して，冗長なデータのロード操作を省き，できる限り再利用するように指定するトグルです。

注意：コンパイラは，レジスタがポインタによって間接的に変更されたような場合を検出できないので，このオプションは注意して使ってください。この制約に関しては，付録 B で詳しく解説しています。

Jump optimization（ジャンプの最適化）:

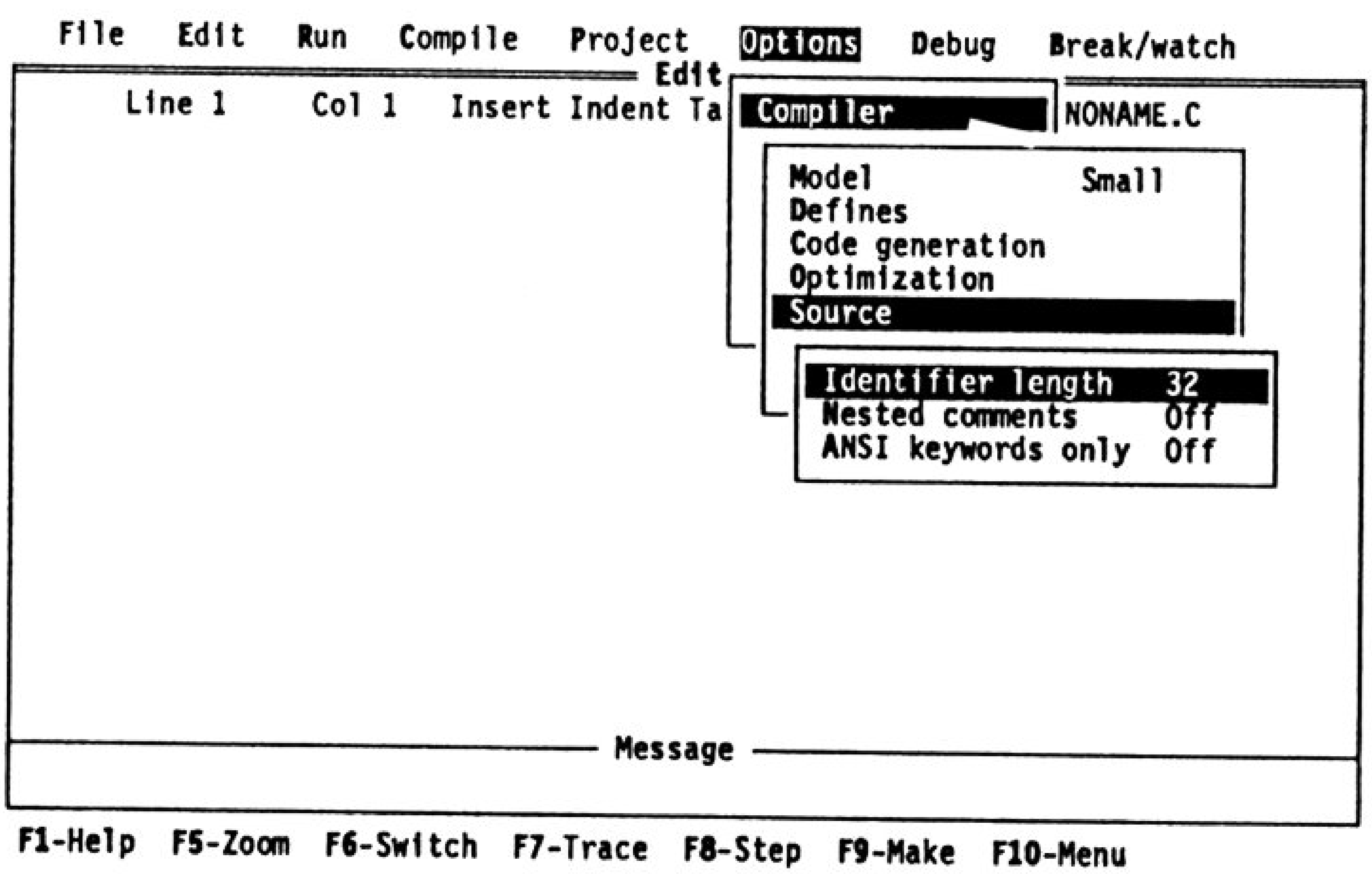
無駄なジャンプ（ジャンプ命令へのジャンプなど）を削除したり，ループや switch 文を再編成したりして，コードサイズを小さくするかどうかを指定します。ループの再編成によって，内部ループが圧縮されスピードアップされます。

注意：このスイッチが On になっていると，複数のソースコード行から1つのコードシーケンスが生成されることがあるので，統合デバッガでのトレースやステップ実行の順序が混乱する場合があります。最良の結果を得るには，デバッグ中はこのスイッチを Off にしておいてください。

Source メニュー（ソースファイル）

このメニューの項目は，コンパイラがコンパイルの初期フェーズでソースファイルをどのように扱うかに関するものです。

図 5.15　Options/Compiler/Source メニュー



Identifier length（**識別名の長さ**）：

識別名の先頭から何文字目までが有効かを指定します。識別名は，先頭から指定された文字数分の範囲が異なっていたときにのみ，別々の識別名であると認識されます。識別名には，変数，プリプロセッサのマクロ名，構造体のメンバ名，などが含まれます。文字数は1～32の範囲で指定することができ，デフォルトは32になっています。

Nested comments（**コメントのネスト－入れ子**）：

Turbo C のソースファイルで，コメントのネストを許すかどうかを指定します。C の多くの処理系ではコメントのネストは許されておらず，可搬性を低くしています。

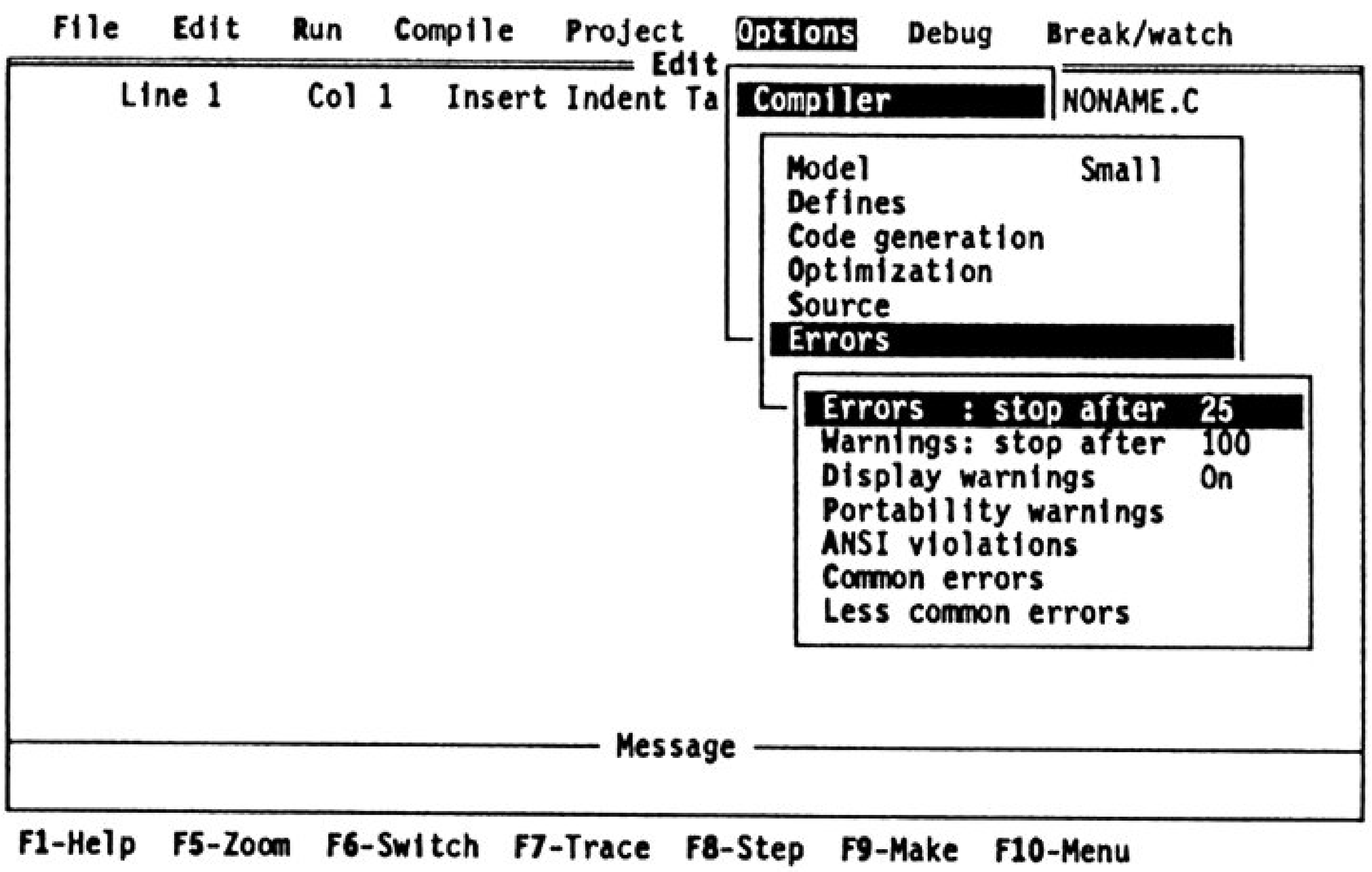
ANSI keywords only（**ANSI の予約語のみ**）：

このトグルが On になっていると，ANSI 標準化案で定義されている予約語だけが予約語として認識され，Turbo C で拡張された予約語は通常の識別名として扱われます。Turbo C で拡張された予約語には，near，far，huge，asm，cdecl，pascal，interrupt，_es，_ds，_cs，_ss，およびレジスタ疑似変数（_AX，_BX，……）が含まれます。また，このオプションが On の場合には，コンパイル時にシンボル __STDC__ が定義されます。

Errors メニュー（エラー）

このメニューからの指定で，Turbo C コンパイラが診断メッセージをどのように扱い，どのように反応するかを制御します。

図 5.16　Options/Compiler/Errors メニュー



Errors　:　stop after（エラーの制限個数）:

エラーを何個検出したらコンパイルを中止するかを指定します。デフォルトは25個になっています。個数は0～255の範囲で指定し，0は無制限を意味します。

Warnings　:　stop after（警告の制限個数）:

警告が何個発生したらコンパイルを中止するかを指定します。デフォルトは100個になっています。個数は0～255の範囲で指定し，0は無制限を意味します。

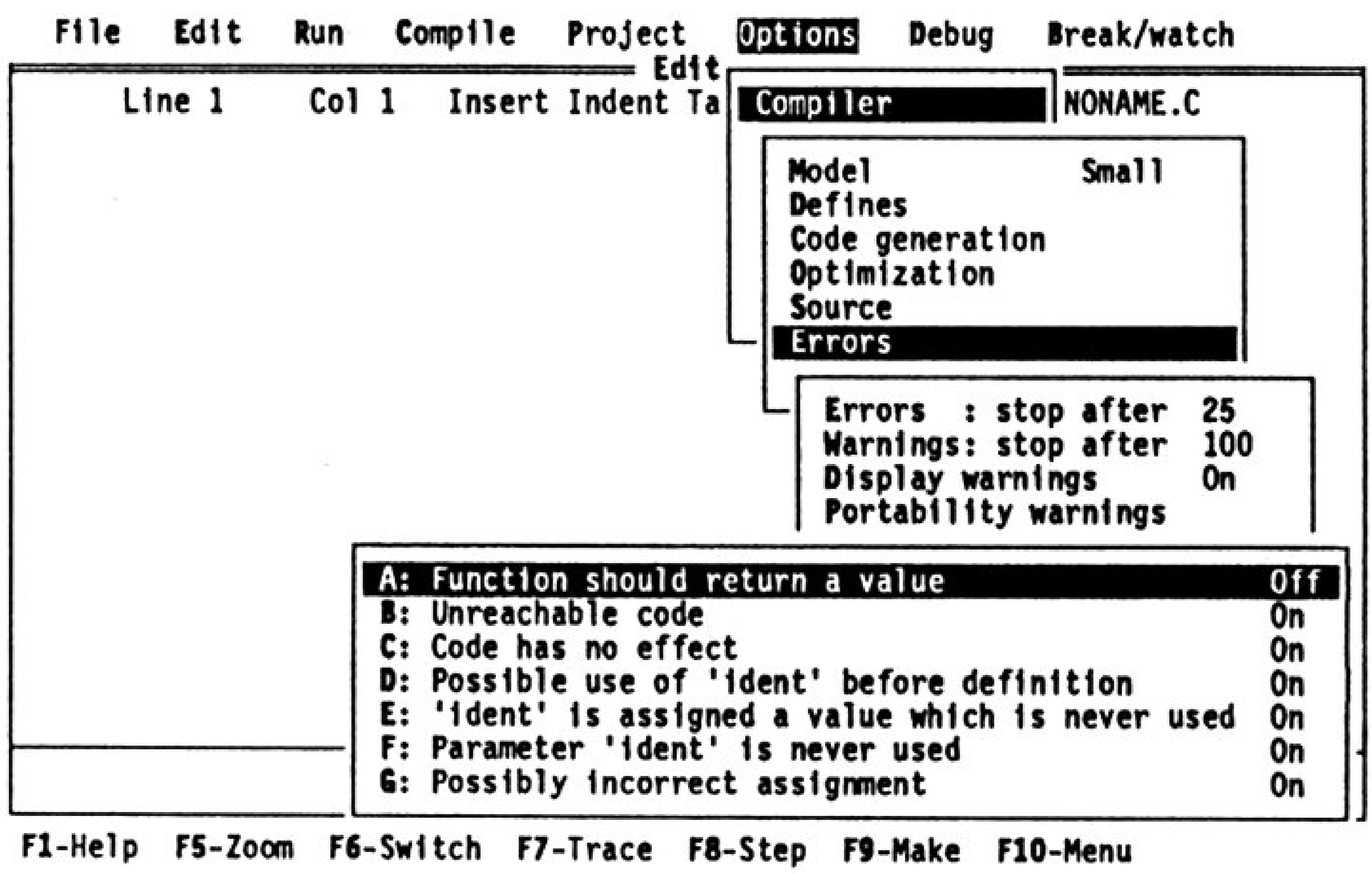
Display warnings（警告の表示）：

警告メッセージを表示するかどうかを指定するトグルです。デフォルトはOn(表示する）で，この場合には以下の4種類の警告についてさらに細かな指定が行なえます。

- Portability warnings（可搬性に関する警告）
- ANSI violation（ANSI 案に対する違反）
- Common errors（一般的な警告）
- Less common errors（それほど一般的でない警告）

この項目が Off になっていると，警告メッセージはいっさい表示されません。警告メッセージの詳細については，付録 B およびリファレンスガイドの付録 A を参照してください。

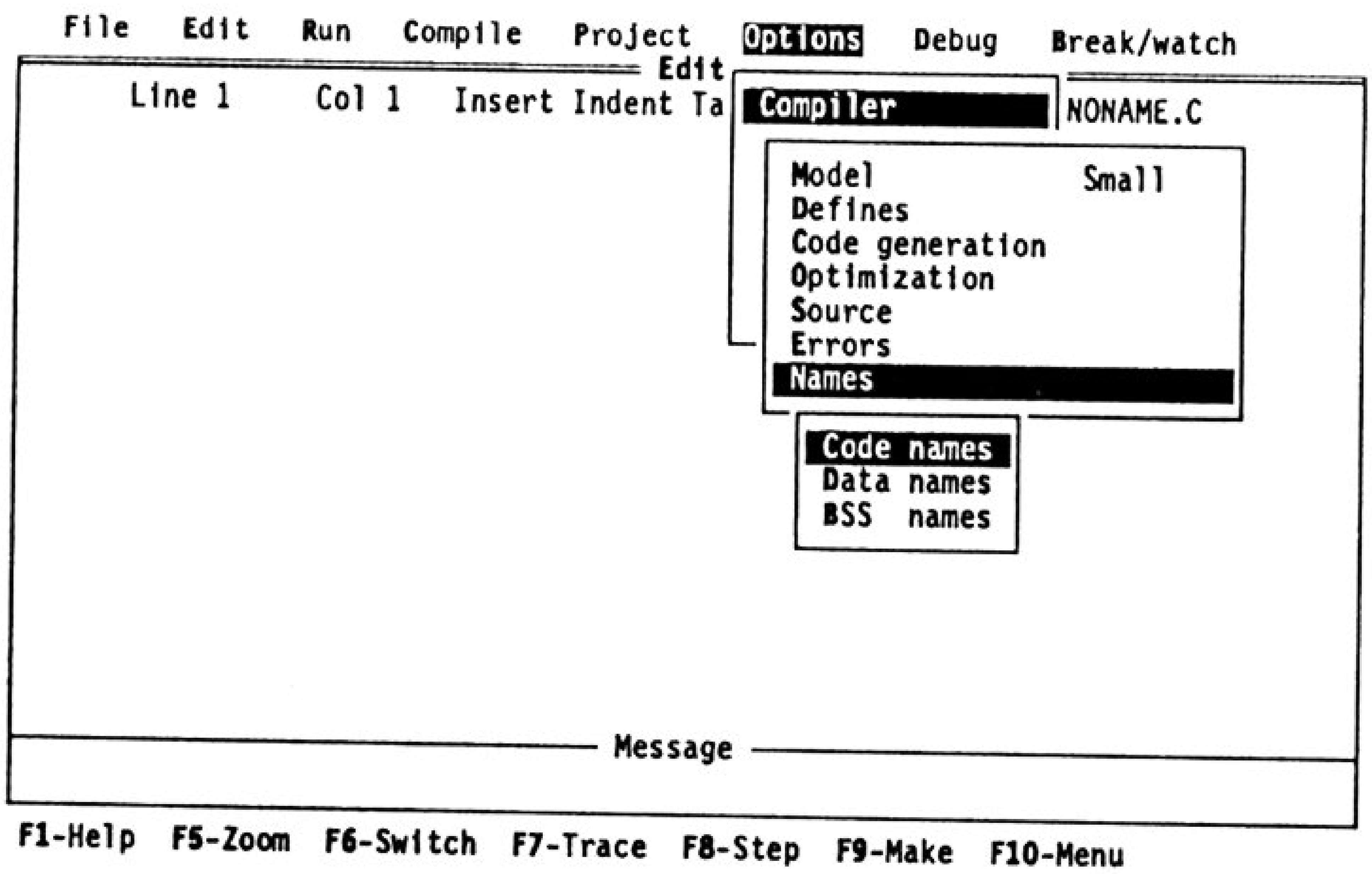
図 5.17　Common errors の表示



Names メニュー（セクションの名前）

このメニューの項目によって，Code，Data，BSSの3つのセクションの，セグメント名，グループ名，およびクラス名のデフォルト値を変更することができます。

図5.18　Options/Compiler/Names メニュー



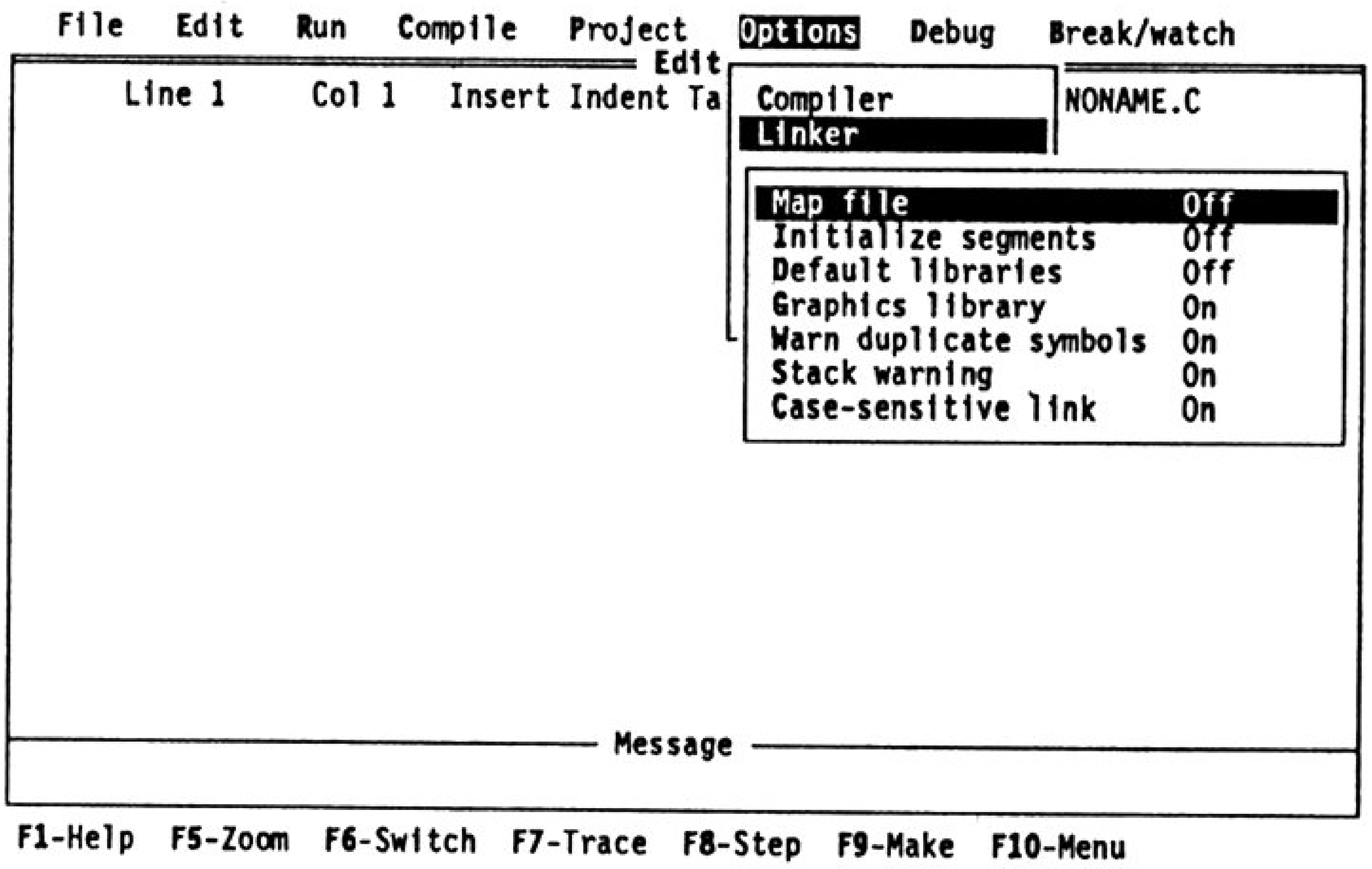
3つの項目からどれかを選ぶと次のメニューが表示されますが，この中のアスタリスク（＊）はデフォルトの名前が使用されることを意味しています。

注意：このオプションは，第12章を読み終えてよく理解された方以外は，変更しないようにしてください。

Linker（リンカ）

このメニューの各項目では，リンカに関するオプションの設定を行ないます。この設定に関する詳細については付録 C の TLINK の節を参照してください。

図 5.19　Options/Linker メニュー



Map file メニュー（マップファイル）

出力するマップファイルの形式を選択します。Off 以外に指定されていれば，**O/D/Output directory** で設定されているディレクトリにマップファイルが出力されます。デフォルトは Off になっていますが，この他に Segments，Publics，または Detailed を選択することができます。

Initialize segments（セグメントの初期化）

初期化されないセグメントを初期化するようにリンカに指示します（通常は初期化する必要はなく，初期化を行なうようにすると.EXE ファイルは必要以上に大きくなることになります）。

Default libraries（デフォルトライブラリ）

Turbo C 以外のコンパイラで作成したモジュールとのリンクを行なうときに，このオプションが必要になる場合があります。コンパイラによっては，オブジェクトファイルの中にデフォルトのライブラリのリストを書き込むようになっているものがあります。

このオプションが On になっていると，リンカは Turbo C のデフォルトライブラリだけでなく，オブジェクトファイル内のリストにあるライブラリも，未定義ルーチンを探す対象とします。

このオプションが Off になっていると，Turbo C のデフォルトライブラリだけが探索され，.OBJ ファイル内のリストにあるライブラリは無視されます。

Graphics libraries（グラフィックスライブラリ）

BGI グラフィックスライブラリを自動的に探すかどうかを指定します。このトグルが On のときには，単一ファイルのグラフィックスプログラムを，プロジェクトファイルなしで作成・実行することができます。このトグルを Off にしておくと，リンカが BGI グラフィックスライブラリファイルをリンクする必要がないので，リンク時間を短縮することができます。デフォルトは On です。

注意：このトグルを Off にセットしていても，プロジェクトファイルの中に BGI グラフィックスライブラリファイルの名前を書いておけば，BGI グラフィックスを使用するプログラムを作成することができます。

Warn duplicate symbols（シンボル重複の警告）

オブジェクトファイルとライブラリファイルの中で，シンボルが重複していた場合に，警告メッセージを出すかどうかを指定します。デフォルトは Off です。

Stack warning（スタックの警告）

リンカが，No stack という警告を出すかどうかを指定します（このトグルが On の場合，タイニィモデルで生成されたプログラムに対しては，通常このメッセージが出力されます）。デフォルトは On（警告を出す）になっています。

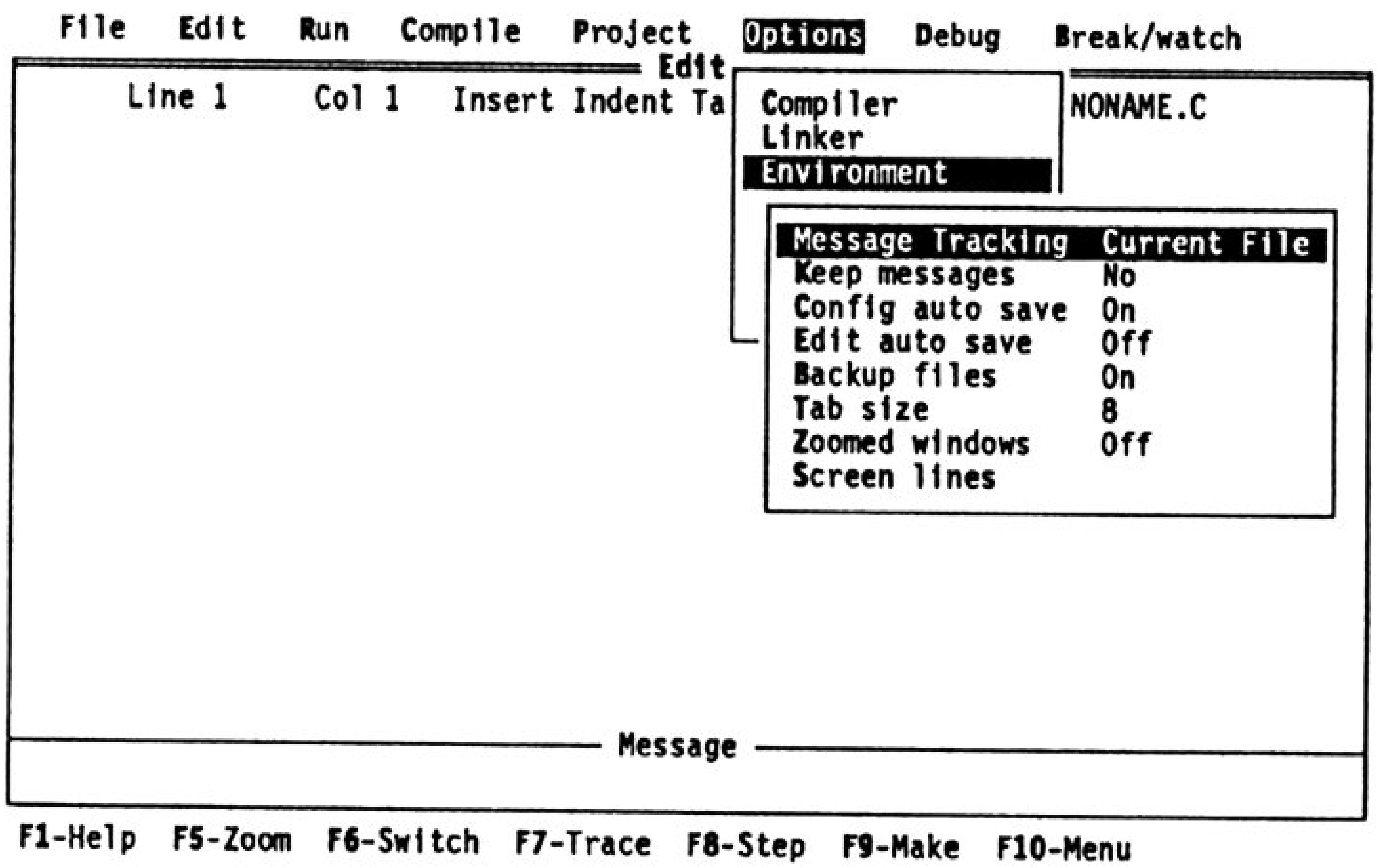
Case-sensitive link（大/小文字の区別）

リンク時に大文字と小文字を区別するかどうかを指定します。Cは大文字/小文字を区別する言語であり，このオプションはデフォルトではOn（区別する）になっています。

Environment メニュー（環境）

このメニューでは，プログラミングの必要に応じて作業環境を変更するオプションがまとめられています。

図 5.20　Options/Environment メニュー



Message tracking（メッセージの追跡）

Turbo Cでは，Messageウィンドウの中のメッセージをスクロールしていくと，ソースファイルの対応する箇所がEditウィンドウに表示されるようになっています。この項目では，その際にどのファイルの中を追跡していくかを，次の3つの中から選択します。

・Current fileがデフォルトで，エディタ内のファイルだけが追跡されます。
・All filesとすると，メッセージの原因となったすべてのファイルが追跡されます。

・Offにすると，追跡はいっさい行なわれません。

Keep messages（メッセージの保持）

これはOnかOffのトグルです。Onの場合には，Turbo CはMessageウィンドウ内のメッセージを保存し，次のコンパイル時のエラーメッセージはその下に追加されます。ファイルをコンパイルするときには，そのファイルに関するメッセージはMessageウィンドウから消去され，新しいメッセージが最後に追加されます。このトグルがOffの場合は，コンパイルやMakeが行なわれる前に，メッセージはすべて自動的にクリアされます。

Config auto save（コンフィギュレーションファイルの自動セーブ）

通常コンフィギュレーションファイルは，**O**ptions/**S**ave optionsコマンドを選択した場合にのみセーブされます（ディスクに書き込まれます）。Config auto saveトグルをOnにしておくと，**R**un/**R**unや**F**ile/**O**S shellを実行するときや，Turbo Cを終了するときに，コンフィギュレーションファイルが自動的にセーブされます（まだセーブされていない場合，あるいは前回セーブしたときから変更が加えられている場合だけです）。

Config auto saveがOnの場合，コンフィギュレーションファイルがまだセーブされていないときには，TCはセーブするファイルの名前を自動的に選択します。このファイルの名前は，前回**O**/**S**ave optionsまたは**O**/**R**etrieve optionsでセーブまたは読み込まれたコンフィギュレーションファイルの名前，あるいは（ロード・読み込み・セーブが行なわれていなければ）カレントディレクトリのTCCONFIG.TCになります。

Edit auto save（編集ファイルの自動セーブ）

このトグルをOnにしておくと，**R**un/**R**unや**F**ile/**O**S sellを実行するときに，エディタ内のソースファイルが自動的にセーブされます（まだセーブされていない場合，あるいは前回セーブしたときから変更が加えられている場合だけです）。

Backup files（ファイルのバックアップ）

デフォルトでは，**F**ile/**S**aveを行なうときには自動的にソースファイルのバックアップが作られます。バックアップファイルは，ファイル名は同じで，拡張子が.BAKに置き換えたものになります。この動作は，このトグルをOffにすれば行なわれなくなります。

Tab size（TAB の桁数）

エディタのタブモードがオンのときに ***TAB*** キーをタイプすると，エディタはファイル内にタブ文字（¥t, 0x09）を書き込み，カーソルは次のタブ位置に移動します。この項目で，タブ位置の間隔を2～16の範囲でを設定することができます。

タブ間隔を変えるには，Tab size を選択して好きな桁数を入力し，**リターン**を押します。するとエディタは，いま設定したタブ間隔に合わせて画面を再表示します。新しいタブ間隔はコンフィギュレーションファイルに保存しておくことができます（Options/Save options を選択してください）。

Zoomed windows（ウィンドウの拡大表示）

Turbo C 統合環境の画面が，Edit ウィンドウと Message ウィンドウの両方が見えるようにセットアップされているときに Zoomed windows を On にすると，アクティブウィンドウが画面いっぱいにズームイン（拡大）されます。***F6*** を使うと，分割画面の場合と同様に，もう一方のウィンドウに切り換えることができます。

ウインドウをズームアウト（拡大）して，2つのウィンドウが両方とも見えるような設定に戻すには，Zoomed windows を Off にセットしてください。

Screen size メニュー（画面サイズ）

Screen size を選ぶと次のメニューが現われ，統合環境の画面表示の形式を選択することができます。メニューの内容は，PC-9801シリーズと IBM PC では異なります。

PC-9801：

Standard 25 line（標準25行），**E**xtended 23 line（拡張23行），**H**igh density 46 line（圧縮46行）の中から選択します。標準モードでは，8色表示しかできませんが，拡張モードと圧縮モードでは，マシンの構成によっては16色の表示が可能になります。なお，拡張モードと圧縮モードは，グラフィック VRAM のバンク#1を持っていないマシン（最初期型の PC-9801や PC-9801U2），および，TCINST の **M**ode for display を Text only に設定している場合には選択できません。画面モードの詳細については付録 D を参照してください。

IBM PC：

25, 43/50行のいずれかを指定します。このうちの1つあるいは2つは，使用しているPCが装備しているビデオアダプタのタイプによって選択できないようになっています。

■ 25 lines

これは，25行×80桁の標準PCディスプレイです。この項目は常に選択可能です。モノクロディスプレイアダプタ（MDA）またはカラーグラフィックスアダプタ（CGA）のシステムでは，このサイズしか選択できません。

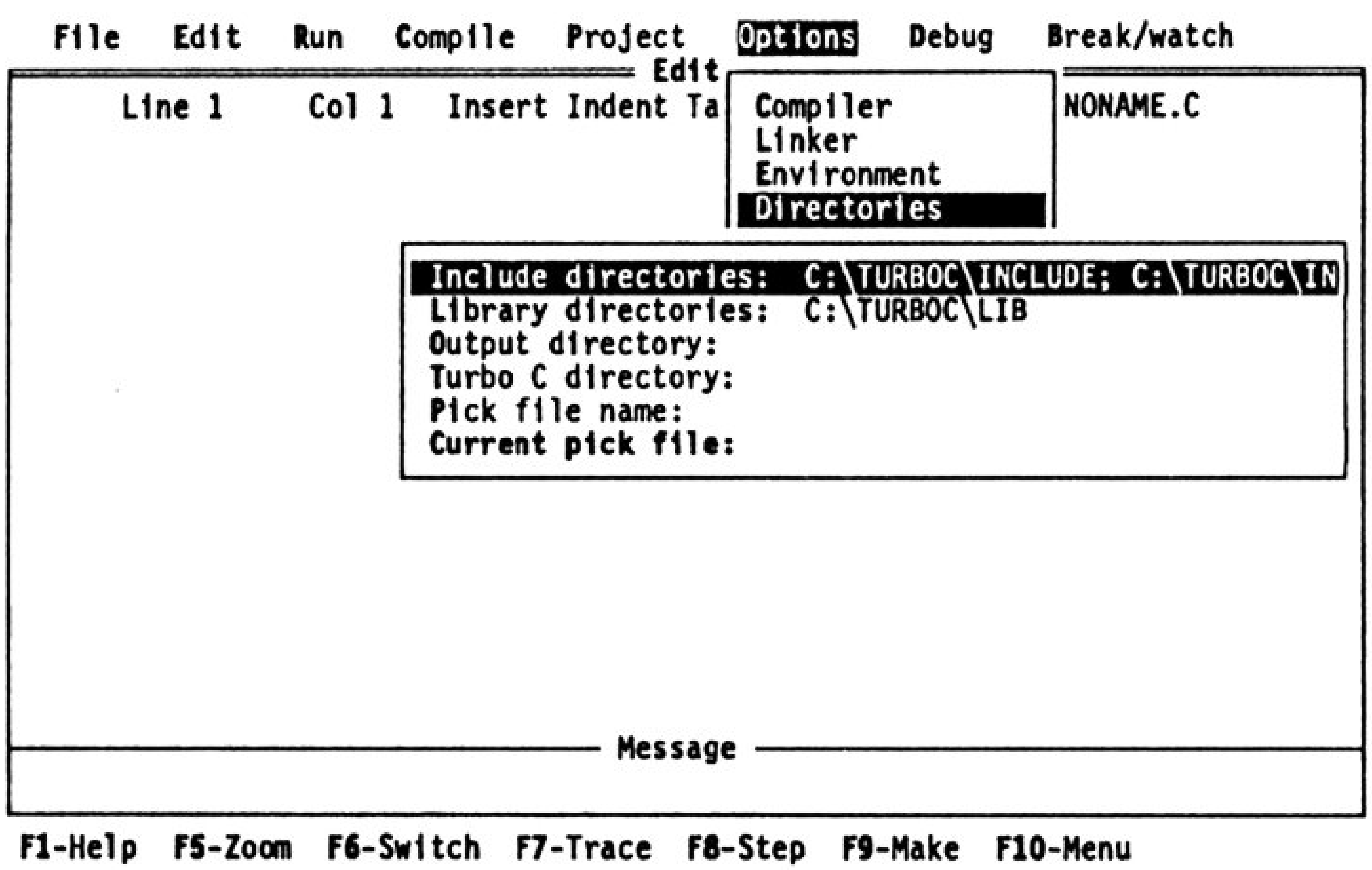
■ 43/50 lines

EGAまたはVGAを装備したPCでは，25行標準ディスプレイに加えて，この項目を選択することができます。EGAであれば43行×80桁，VGAであれば50行×80桁でテキストが表示されます。

Directories（ディレクトリ）

このメニューでは，コンパイルやリンク，ヘルプメッセージの表示などに必要なファイルがどこに置かれているかを指定します。また，プログラミングの必要に応じて作業環境を変更するオプションもいくつか含まれています。

図 5.21　Options/Directories メニュー



Include directories（インクルードディレクトリ）

標準インクルードファイルが置かれているディレクトリを指定します。標準インクルードファイルとは，ソースコード中の# include 文で不等号記号〈〉で囲んで与えられるものです（**例**:# include 〈myfile.h〉)。複数のディレクトリを指定する場合には，セミコロン（;）で区切ってください。指定できるのは，空白文字なども含めて最大127文字までです。セミコロンの前後には，ホワイトスペース（スペースまたはタブ）があってもなくてもかまいません。このオプションについては第3章で詳しく説明しています。

Library directories（ライブラリディレクトリ）

Turbo C のスタートアップオブジェクトファイル（C0?.OBJ）および実行時ライブラリルーチン（.LIB ファイル）が置かれているディレクトリを指定します。

ライブラリディレクトリを入力するときは，以下に示すガイドラインにしたがってください。

■複数のライブラリディレクトリは，セミコロンで区切らなければなりません。
■セミコロンの前後にホワイトスペース（スペースまたはタブ）を置いてもかまいません（なくてもかまいません)。
■絶対パス・相対パスの両方が使用できます。カレントディレクトリからの相対パスだけでなく，他のドライブ上のログドポジションからの相対パスも使うことができます。

例を示します。

```
A:¥TURBOC¥LIB; B:¥TURBOC¥MYLIBS; A:NEWTURBO¥MATHLIBS; A:..¥VIDLIBS
```

Output directory（出力ディレクトリ）

.OBJ,.EXE,.MAP ファイルが出力されるディレクトリです。Make や Run を実行するときには，ここで指定されているディレクトリが探索されます。この指定が空白になっていると，これらのファイルはカレントディレクトリに出力されます。

Turbo C directory（Turbo C ディレクトリ）

これは，Turbo C システムが，コンフィギュレーションファイル（.TC）とヘルプファイル（TCHELP.TCH）を見つけるために使用します。デフォルトのコンフィギュレーションファイルをカレントディレクトリとは別の場所に置いている場合に，Turbo C の起動時にそれを読み込ませるようにするには，TCINST（カスタマイズプログラム）を使ってそのパスを設定しておかなければなりません。

Pick file name（ピックファイル名）

この項目では，ロードするピックファイルの名前を定義します(デフォルトは TCPICK.TCP です)。ここで名前を入力すると，そのピックファイルが（もしあれば）ロードされ，また Turbo C を終了するときにセーブされるピックファイルもこの名前になります。ピックファイル名変更すると，新しいピックファイルをロードする前に，それまでのピックファイルはセーブされます。

ここでピックファイル名が指定されていない場合は，Turbo C は Current pick file に設定されているファイル名でのみ書き込みを行ないます。

ピックファイルを新たに作る場合は，このオプションによってピックファイル名を定義しなければなりません。ピックファイル名が定義されていれば，Turbo C は統合環境を終了するときには必らずディスク上のそのピックファイルが更新されます。ピックファイル名は，Options/Save options を選択するとコンフィギュレーションファイルにセーブされます。

Current pick file（現在のピックファイル）

このメニュー項目は，ピックファイル名とその置かれている場所を（もしあれば）表示します。この項目は情報を表示するだけのもので，なにかを指定することはできません。デフォルトのピックファイルがロードされているとき，あるいは Pick file name メニューでピックファイルを入力したときに，Current pick file にそのファイル名が表示されます。ピックファイルの名前を変更した場合や，統合環境を終了するときには，現在のピックリストの情報がこのファイルに格納されます。

Arguments（引数）

ここでは，実行するプログラムに，MS-DOS のプロンプトからの場合と同様なコマンドライン引数を与えることができます(ただしリダイレクションはできません)。指定するのは引数のみで，プログラム名は必要ありません。

Save options（オプションの保存）

O/Compiler，**O**/**L**inker，**O**/**E**nvironment，**O**/**D**irectories，**D**ebug，**P**roject で選択されているすべてのオプションを，コンフィギュレーションファイル（デフォルトのファイルは TCCONFIG.TC）にセーブします。Turbo C は，起動時にカレントディレクトリの中から TCCONFIG.TC を探します。もし見つからなければ，Turbo C ディレクトリの中を探します。

Retrieve options（オプションの読み込み）

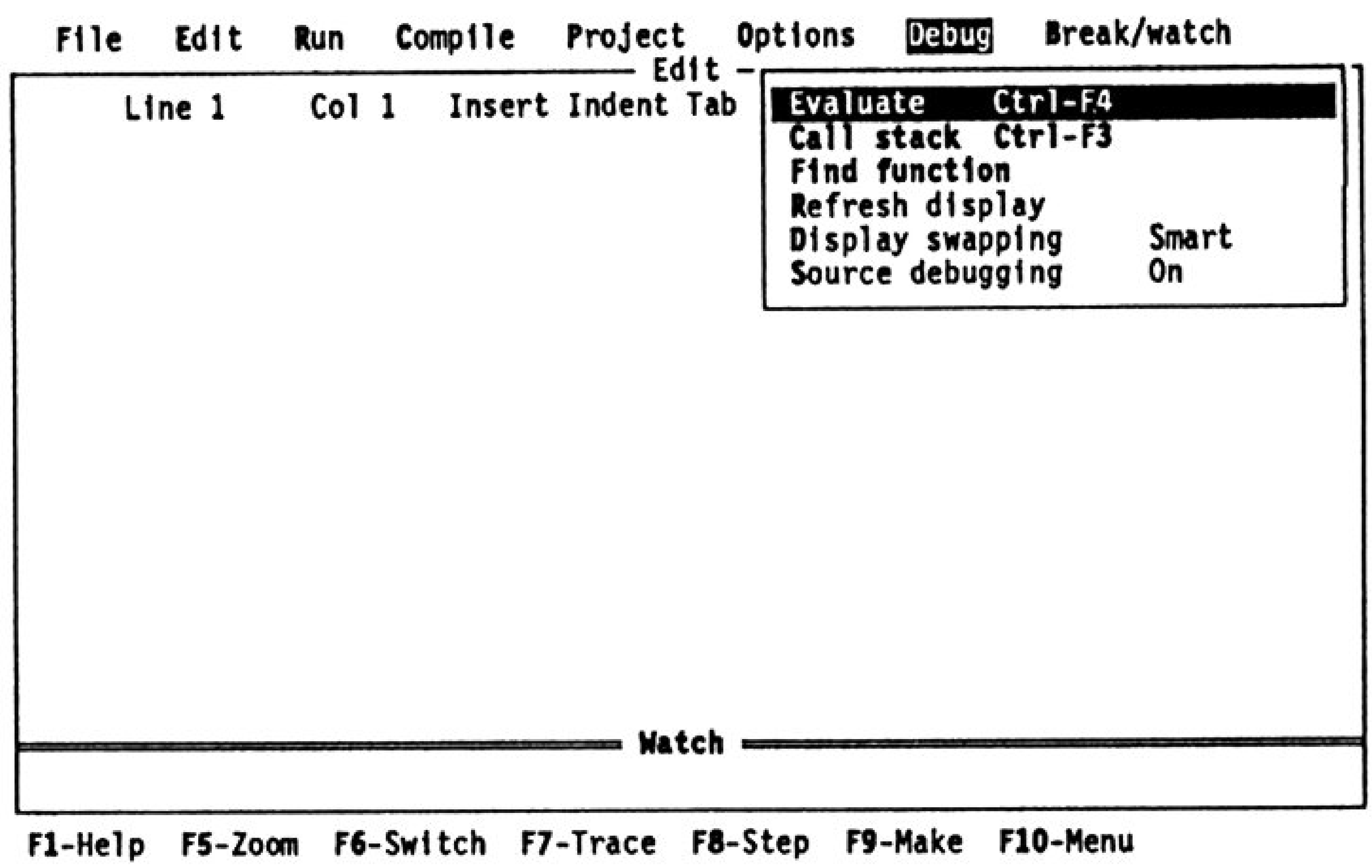
Save options コマンドでセーブされたコンフィギュレーションファイル，あるいは指定のコンフィギュレーションファイルを読み込みます。コンフィギュレーションファイルは，名前を変えていくつでも作っておくことができ，**R**etrieve options を選択することによって，必要に応じてコンフィギュレーションファイルを切り換えることができます。コンフィギュレーションファイルについては第 **III** 部で詳しく説明しています。

Debug メニュー

Debug メニューのコマンドは，ブレークポイントと監視式を除く統合デバッガの機能をコントロールします（ブレークポイントと監視式については Break/watch でコントロールします）。

デバッガの操作手順などについては第4章で取り上げています。

図 5.22　Debug メニュー



Evaluate（式の評価）

Evaluate は変数や式を評価してその値を表示します。適切であればその値を変えることもできます。

このコマンドを選択すると，Evaluate（評価），Result（結果），New value（新規値）の3つのフィールドを持つポップアップウィンドウが開かれます。ウィンドウが開かれたときに，Evaluate フィールドには，Edit ウィンドウのカーソル位置の語がデフォルト式として入力されます。**リターン**キーを押すと，このデフォルト式を評価することができます。また，この式は編集したり別の式に置き換えたりすることができます。右矢印キーを押すと，Edit ウィンドウからさらに文字がコピーされ，デフォルト式を拡張することができます。

次に示したものが含まれていなければ，どんなCの式でも評価することができます。

■関数呼び出し
■# define または typedef で定義されたシンボルあるいはマクロ
■実行中の関数のスコープに入っていないローカル変数あるいは静的変数で，完全には修飾されていないもの

デバッガがその式を評価できる場合には，その値を Result フィールドに表示します。

式が1つの変数あるいは単純データ要素を参照するものであれば，カーソルを New value フィールドに移動して，新しい値として式を入力することができます。

式の値を修正することが有意であっても，そうしたくないのであれば，***ESC*** を押してウィンドウを閉じてください。New value フィールドの内容を変更していても，**リターンキー**を押さずにウィンドウから出てしまうと，デバッガはその変更を無視します。

Debug/Evaluate は，それぞれの型の値を適切なフォーマットで表示します。たとえば，int 型は10進整数として，配列はポインタとして16進で表示します。別のフォーマットで表示するには，式の後にカンマとフォーマット指定子をつけてください。フォーマット指定子は**表 5.5** に示されています。

連続したデータ要素の値を表示するには，繰り返し回数を用いてください。たとえば，*xarray* という整数型配列があったとすると，次のようにします。

xarray[0],5	連続した5つの整数を10進で表示します。
xarray[0],5x	連続した5つの整数を16進で表示します。

繰り返し回数とともに使用できる式は，単一データ要素を表わすものでなければなりません。デバッガは，データ要素がポインタでなければそれを配列の最初の要素とみなし，それがポインタであれば配列へのポインタとみなします。

表 5.5　デバッガの式で使用できるフォーマット指定子

文字	機能
C	**文字**。制御文字（ASCII コードの0から31）に対して特別な表示文字を出します。たとえば，^Aは，"ニコニコ顔"として表示されます。これは文字や文字列に対して働きます。

S　**文字列**。制御文字（ASCII コードの0から31）を C 言語のエスケープ文字列を使用して ASCII コードで表示します。これは，文字や文字列の表示に対するデフォルトのフォーマット指定子であり，M 指定子と組み合わされて使用される場合にのみ役に立ちます。

D　**10 進**。すべての整数値は，10進で表示されます。単純な整数型の式の他に，整数を含む配列や構造体に対しても働きます。

H または X　**16 進**。すべての整数値は，0x を前につけて16進で表示されます。単純な整数型の式の他に，整数を含む配列や構造体に対しても働きます。

F〈*n*〉　**浮動小数点**。*n* は2から18の間の整数で，表示する有効数字の桁数を示します。浮動小数点値にのみ働きます。

M　**メモリダンプ**。指定された式のアドレスからのメモリダンプを表示します。式は代入文の左辺における構文，つまりメモリアドレスを示すものでなければなりません。そうでない場合 M 指定子は無視されます。デフォルトでは，変数の各バイトは2桁の16進数で表示されます。M 指定子に D 指定子をつけると各バイトは10進で表わされ，H 指定子または X 指定子をつけると各バイトは16進で表わされます。C 指定子または S 指定子が指定されると，変数は（特殊文字を含むものも含まないものも）文字列として表示されます。表示されるデフォルトのバイト数は変数のサイズに対応しますが，繰り返し回数を使って指定してもかまいません。

P　**ポインタ**。ハードウェア志向のデフォルトの *seg*：*ofs* 型のフォーマットだけではなく，指されているアドレスについての情報を合わせ持った *seg*：*ofs* 型のフォーマットでポインタを表示します。特に，セグメントが置かれているメモリ領域とオフセットアドレスに存在する変数名の名前を示します。メモリ領域を次に示します。

メモリ領域	Evaluate メッセージ
0000：0000 － 0000：03FF	Interrupt vector table （割り込みベクターテーブル）
0000：0400 － 0000：04FF	BIOS Data area（BIOS データ領域）

0000：0500 - Turbo C	MSDOS / TSR's (MS-DOSとメモリ常駐プログラム)
Turbo C - ユーザプログラム PSP	Turbo C
ユーザプログラム PSP	User Process PSP (ユーザプロセスの PSP)
ユーザプログラム-RAMの最後部	変数に割り当られたメモリ内にそのアドレスがあれば静的ユーザ変数の名前，そうでなければ何もなし

■ **PC-9801の場合**：

A000：0000 - A000：3FFF	Text Video RAM(テキスト VRAM)
A000：4000 - A000：7FFF	Not Used (未使用)
A800：0000 - B000：FFFF	Graphic Video RAM (グラフィック VRAM)
C000：0000 - C000：7FFF	User Enhanced ROM (ユーザ拡張 ROM)
C800：0000 - E000：FFFF	System enhanced ROM (システム拡張 ROM)
E000：0000 - E000：7FFF	Graphic Video RAM (グラフィック VRAM)
E800：0000 - F000：FFFF	N88-BASIC RAM

■ **IBM PCの場合**：

A000：0000 - A000：FFFF	EGA Video RAM(EGAビデオRAM)
B000：0000 - B000：7FFF	Monochrome Display RAM (モノクロディスプレイ RAM)
B800：0000 - B800：7FFF	Color Display RAM (カラーディスプレイ RAM)
C000：0000 - E000：FFFF	EMS Pages / Adapter BIOS ROM's (EMSページ/アダプタ BIOS ROM)
F000：0000 - F000：FFFF	BIOS ROM

R **構造体/共用体**。{ X:1, Y:10, Z:5 }のように値とフィールド名を表示します。構造体と共用体に対してのみ働きます。

Debug/Evaluateのホットキーは ***CTRL-F4*** です。

Find function（関数の検索）

Find function は，Edit ウィンドウに関数の定義部を表示します。このコマンドで見つけられる関数は，**Debug/Source debugging** オプションと **O/C/C/OBJ debug information** オプションを On にしてコンパイルしたプログラム内で定義されており，そのソースファイルが参照可能なものです。探索する関数が現在表示されているファイルの中にない場合は，適切なファイルが自動的にロードされます。

Find function を使うためには，デバッグセッションに入っていなければなりません。

Call stack（コールスタック）

Call stack は，コールスタックを含んでいるポップアップウィンドウを表示します。コールスタックとは，現在実行中の関数に至るまでの道筋で呼び出された関数の並びを表わします。**main** はスタックの底にあり，スタックトップ（一番上）には現在実行中の関数があります。

コールスタック上の各エントリには，呼び出された関数名と渡されたパラメータの値が表示されます。

最初はスタックトップの要素がハイライト表示されます。コールスタック上の他の関数の現在行を表示するには，ハイライトをその関数名のところに移して**リターン**キーを押してください。すると，その関数内で，スタック上のすぐ上にある関数を呼び出している行にカーソルが置かれます。たとえば，コールスタックが次のようなっているとします。

```
func2()
func1()
main()
```

（これは **main** が *func1* を呼び出し，*func1* が *func2* を呼び出していることを意味しています）。ここで，*func1* の現在実行している行を見たい場合は，ハイライトバーをコールスタック上の *func1* に動かして**リターン**を押してください。*func1* のコードが Edit ウィンドウに現われ，カーソルが *func2* を呼び出している位置に置かれます。

現在実行中の関数の現在行（実行位置）に戻りたいときは，コールスタックの先頭にハイライトバーを動かして**リターン**を押してください。

O/C/C/Standard stack frame オプションを Off にしてコンパイルしたプログラムの場合，いくつかの関数がコールスタックから省かれることがあります。詳しくは **O/C/C/Standard stack frame** の説明を参照してください。

D/Call stack のホットキーは ***CTRL-F3*** です。

Refresh display（画面の再表示）

エディタ画面がなんらかの障害によって上書きされてしまった場合は，このオプションを使って前の内容を復元することができます。

Display swapping（画面の切り換え）

Debug/Display swappingは，Smart，Always，Noneの3つのどれかを選択するトグルです。デフォルトはSmartです。

デフォルトのSmartの設定で，プログラムをデバッグモードで実行すると，デバッガは実行中のコードが画面出力をするかどうかを調べます。そのコードが画面出力をする（あるいは関数を呼び出す）ものであれば，画面は出力が行なわれている間Edit画面からユーザ画面に切り換わって，出力が終了したらもとに戻ります。その他の場合は，画面の切り換えは行ないません。

注意：Smartは，次の点に関してはとりたてて賢い（smart）わけではありません。

■関数が画面出力をしない場合でも，関数呼び出しがあると画面は切り換えられます。
■ある状況下——たとえば，タイマー割り込みルーチンが画面に書き込むような場合には，画面の切り換えがなされずに，エディタ画面が変更を受けることがあります。

Alwaysにすると，1つの文を実行するたびに画面が切り換わります。プログラムの実行によりエディタ画面が上書きされそうな場合には，これを選択するのがよいでしょう。

Noneにすると，デバッガは画面の切り換えをまったく行ないません。画面出力がないことが確かなコードに対するデバッグセッションでは，これを選択するのがよいでしょう。

注意：デュアルモニタモード（TCコマンドラインの/dスイッチを使った場合）でデバッグをしているときは，一方のモニタでプログラムの出力を見て，もう一方のモニタでTCの画面を見ることができます。このためにTCは画面の切り換えを行なわないので，Debug/Display swappingの設定は意味がなくなります。

Source debugging

Debug/Source debuggingオプションは，On，Standalone，Noneの3つからどれかを選

ぶトグルです。

On に設定してリンクしたプログラムは，Turbo C 統合デバッガあるいはスタンドアロンの Turbo デバッガの両方でデバッグすることができます。

Standalone の場合は，Turbo デバッガでのみプログラムをデバッグすることができます。しかし，この場合でも Turbo C の中でそのプログラムを実行することはできます。

None の場合は，デバッグ情報が.EXE フィルに書き込まれないので，どちらのデバッガでもプログラムをデバッグすることはできません。

Break/watch メニュー

Break/watch メニューコマンドは，ブレークポイントと監視式をコントロールします。

ブレークポイントとは，実行が停止するプログラム内の場所のことで，キーとなる重要な変数や式の値を調べたり，プログラムが正しく動作しているかどうかを確かめるための時間をとるために設定するものです。

ブレークポイントはハイライト表示されます。プログラムがブレークポイントで停止しているときには，そのブレークポイントのハイライトは実行バーによって隠されてしまいますが，実行バーが移動すれば再び現われます。

監視式とは，Watch ウィンドウにその値が表示される式のことで，プログラムが停止したときには必らず再評価されます。有効な監視式を入力する上での規則は，**Debug/Evaluate** の場合の規則と同じです。ただし，監視式の場合は，*i*＋＋のような副作用を生じるものであってはいけません。フォーマット指定子と繰り返し回数は，**Debug/Evaluate** での場合と同様に，監視式の中でも使用することができます。たとえば次の監視式は，

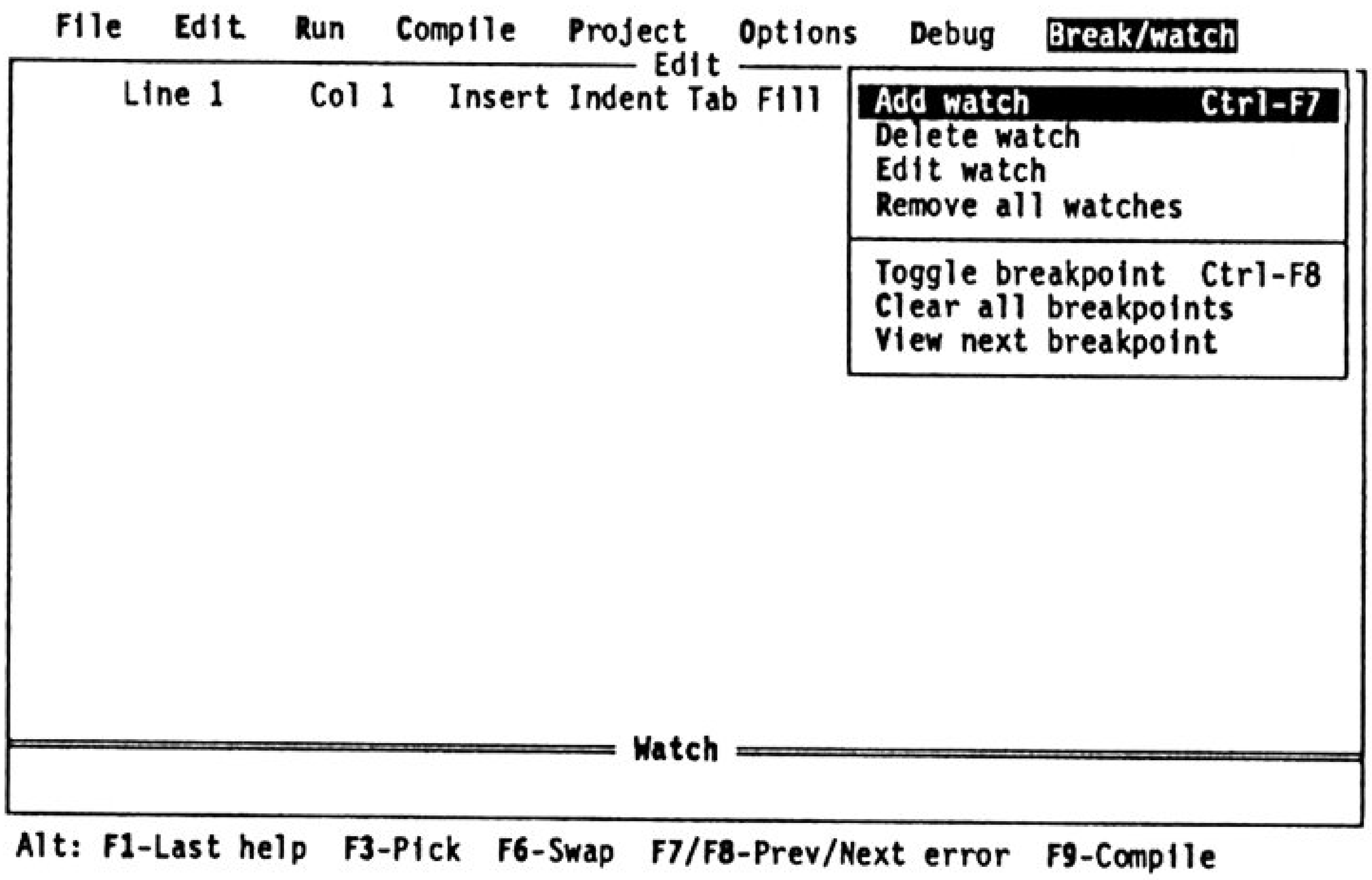
```
i,x
```

整数 *i* の内容を16進のフォーマットで表示します。

Watch ウィンドウに式を追加していくと，ウィンドウは TCINST ユーティリティの **R**esize windows オプションで指定された最大サイズに達するまで拡大していきます（最大サイズの初期値は画面の約半分の大きさです)。最大サイズに達すると，式を追加した場合，いくつかの式がスクロールしてウィンドウから消えてしまいます。それらは，***ROLLUP***，***ROLLDOWN***，上/下矢印によって Watch ウィンドウをスクロールさせれば見ることができます。

Watch ウィンドウの中で現在注目されている式は，Watch ウィンドウがアクティブな場合にはハイライトバーによって示され，アクティブでない場合は左端にマークがつけられます。

図 5.23　Break/watch メニュー



Add Watch（監視式の追加）

Add watch は，Watch ウィンドウに監視式を1つ追加します。このコマンドを選択すると，デバッガはポップアップウィンドウを開いて監視式の入力をうながします。デフォルトの式は，Edit ウィンドウのカーソル位置にある語です。式をタイプして**リターン**キーを押すと，デバッガはその式とその現在の値を Watch ウィンドウに追加します。

このコマンドのホットキーは ***CTRL-F7*** です。また，Watch ウィンドウがアクティブであれば，***INS*** または ***CTRL-N*** を押すことにより新しい監視式を挿入することができます。

Delete watch（監視式の削除）

Delete watch は，Watch ウィンドウから現在注目されている監視式を削除します。

このコマンドを使用するためには，Watch ウィンドウが見えている状態でなければなりません（つまり，Edit ウィンドウが拡大表示されていてはいけません）。現在注目されている監視式は，Watch ウィンドウにいる場合はハイライトバーによって示され，Edit ウィンドウにいる場合は左端のマークによって示されます。

Edit ウィンドウにいるときにマークのついている監視式を削除するには，Break/watch/Delete watch を選んでください。現在注目されている監視式以外の式を削除する

には，Watch ウィンドウに移動して，ハイライトバーを望みの監視式のところに移動させてから，***DEL*** または ***CTRL-Y*** を押してください。

Edit watch（監視式の編集）

Edit watch は，現在注目されている監視式を Watch ウィンドウで編集します。

Break/watch/Edit watch を選択すると，デバッガはポップアップメニューを開いて，そこに現在注目されている監視式をコピーします。式を編集して**リターン**キーを押してください。デバッガは，元の式を編集された式に置き換えます。Watch ウィンドウからも監視式の編集をすることができます。この場合は，ハイライトバーを編集したい監視式のところに移動して**リターン**キーを押してください。

Remove all watches（すべての監視式の消去）

Remove all watches は，Watch ウィンドウからすべての監視式を削除します。

Toggle breakpoint

Toggle breakpoint は，カーソル位置の行にブレークポイントを設定あるいは解除を行ないます。ブレークポイントが設定されると，その行はハイライト表示されます。

このコマンドのホットキーは ***CTRL-F8*** です。

プログラムは，実行途中でブレークポイントに出会うと実行を停止します。プログラムが停止したときには，実行バーはブレークポイントの設定されている行に位置しています。ブレークポイントのハイライト表示は実行バーによって隠れてしまいますが，実行バーが移動すると再び現われます。

ソースファイルの編集中は，各ブレークポイントはそれが設定されている行に“貼りついて”います。これが消えるのは，統合環境から離れたときか，それが設定されているソース行が削除されたとき，あるいは Break/watch/Toggle breakpoint コマンドまたは Break/watch/Clear all breakpoints コマンドでそれが解除されたときに限られます。

Turbo C は次の2つの場合に，ブレークポイントを“見失って”しまいます。

■ブレークポイントを含んでいるファイルを編集してから，編集後のファイルを捨ててしまった場合（Turbo C はファイルが編集される前のブレークポイントの位置を記憶していることができません。そのため，間違った行にブレークポイントを表示してしまいます）。

■ブレークポイントを含んでいるファイルを修正してから，そのプログラムを Make し直さずにデバッグセッションを再開した場合（Turbo C は Source modefied, rebuild? という警告を出します）。

ソースファイルをコンパイルする前には，空白行やコメント行に対してもブイクポイントを設定することができますが，そのファイルをコンパイル・実行すると，Turbo C は設定されたブレークポイントが有効であるかどうかを調べ，無効なブレークポイントについてはユーザに対して削除，無視，変更のいずれかを行なうよう聞いて来ます。ファイルをデバッグしているときは，Turbo C はどの行が実行文を含んでいるかを知っており，無効なブレークポイントを設定しようとすると警告を出します。

Clear all breakpoints

Clear all breakpoints は，プログラムのすべてのブレークポイントを解除します。

View next breakpoint

View next breakpoint は，プログラムの次のブレークポイントにカーソルを移動させます。カーソルは，ブレークポイントが設定された順番にしたがって次のブレークポイントに移動するのであり，プログラムがブレークポイントに出会う順番で移動するのではないことに注意してください。このコマンドはコードを実行するのではなく，Edit ウィンドウでアクティブなブレークポイントの位置を示すためだけのものです。

第 III 部：コンフィギュレーションファイルとピックファイル

コンフィギュレーションファイルとはなにか？

コンフィギュレーションファイルは，基本的には Turbo C に関する情報が入っているファイルです。ここには，選択したコンパイラオプションやリンカオプション，コンパイルやリンクの際に Turbo C がファイルを探すさまざまなディレクトリなどの情報をしまっておきます。

コンフィギュレーションファイルには2つの種類があります。1つは TC.EXE（統合環境版），もう1つは TCC.EXE（コマンドライン版コンパイラ）で使用します。コマンドライン用のコンフィギュレーションファイルはただ1つだけで，TURBOC.CFG という名前でなければなりません。統合環境のコンフィギュレーションファイルには任意の名前をつけることができます。TCCONFIG.TC というファイルが統合開発環境のデフォルトのコンフィギュレーションファイルです。

この節では，統合環境用のコンフィギュレーションファイルについて説明します。コマンドライン用のコンフィギュレーションファイルの使い方については，第3章の「**TURBOC.CFG**」の節を参照してください。

TC のコンフィギュレーションファイル

Turbo C 統合環境に初めて入ったときには，コンフィギュレーションファイルは存在しません。起動時に TC.EXE は，すべてのメニュー項目をデフォルトにセットします（**O**ptions/**C**ompiler/**M**emory model...Small, **O**ptions/**C**ompiler/**C**ode generation/**C**alling convention...C, **O**ptions/**E**nvironment/**K**eep messages...No, など）。そして，統合環境を使っていくうちに，メニュー項目の設定をいくつか変更することになるでしょう。

新しく設定した内容をコンフィギュレーションファイルにセーブせずに Turbo C を終了すると，次に統合環境を起動したときにも，メニュー項目はすべて元のデフォルトのままになります。しかし，設定をコンフィギュレーションファイルにセーブしておけば，次に起動したときに，メニュー項目は選択した値に設定され，再度設定する必要がなくなります。

TCCONFIG.TC

TC.EXEは，起動時にTCCONFIG.TCという名前のコンフィギュレーションファイルを探します。TC.EXEはこのファイルを適当な場所で探します（具体的にどこを探すかについてはあとで詳しく説明します）。TCCONFIG.TCが見つからなければ，統合環境はTC.EXEに組み込まれているデフォルトの設定を使うことになります。

他のTCコンフィギュレーションファイル

DOSのプロンプトから特定のコンフィギュレーションファイルを指定してTC.EXEを起動することもできます。これには/cスイッチを使います（この章の第I部の最初の方にある「**TCコマンドラインスイッチ**」を参照してください）。たとえば，DOSのプロンプトから次のようにタイプすると，

```
tc /cmyconfig
```

Turbo CはカレントディレクトリからMYCONFIG.TCというコンフィギュレーションファイルを探します（拡張子の指定がない場合は.TCを付加します）。

指定されたコンフィギュレーションファイルが見つからない場合は，それを示す警告メッセージを出します。コンフィギュレーションファイルが見つからなくても，統合環境はデフォルトの設定を使ってスタートします。

TCコンフィギュレーションファイルの内容

TCコンフィギュレーションファイルにしまわれる情報は，コンパイラ・リンカのオプションとTC.EXE固有のデータの2つのカテゴリにわけることができます。

コンパイラ・リンカオプションは，もちろんコンパイラおよびリンカを制御するもので，コマンドライン版（TCC）にもこのすべてに対応するオプションが存在します。TC.EXEに固有のデータというのは，統合環境自体に関するものです。これには，**P**roject/**P**roject name，**O**ptions/**D**irectories/**P**ick file name，**O**ptions/**E**nvironmentメニューオプションなどがあります。

コンフィギュレーションファイルを作成する

TCコンフィギュレーションファイルはどうやって作成するのでしょうか?コマンドラ

イン用のコンフィギュレーションファイル(TURBOC.CFG)とはちがって，統合環境用はエディタでは作成することができません。TCコンフィギュレーションファイルは，**Options**メニューの**Save options**を使って作成します。

また，**Options/Environment/Config auto save**を**On**にセットしておけば，統合環境から出るときには必らずデフォルトのTCコンフィギュレーションファイル（TCCONFIG.TC）にその時点での設定がセーブされます。

コンフィギュレーションファイルを切り換える

統合環境の中から，現在のコンフィギュレーションファイルを簡単に他の.TCファイルに切り換えることができます。次のようにします。

- **Options**メニューから**Retrieve options**を選びます。この項目を選ぶとConfig Fileボックスが現われ，前回指定したコンフィギュレーションファイルの名前（初めてのときはデフォルトの*.TC）が表示されます。
- マスク（*.tcや??config.*など）をタイプして**リターン**を押すと,.TCファイルのディレクトリリストが現われ，この中からファイルを選択することができます。特定のファイル名をそのままタイプしてもかまいません（**リターン**を押すとそのファイルがロードされます)。

TC.EXEはTCCONFIG.TCをどこで探すのか？

TC.EXEがデフォルトのコンフィギュレーションファイル（TCCONFIG.TC）を探す場所は2つあります。まず最初はカレント（デフォルト）ディレクトリを探します。カレントディレクトリにTCCONFIG.TCが見つからない場合，TCINSTを使ってTurbo Cディレクトリがすでに設定されていれば，次にTurbo Cディレクトリの中を探します。

Turbo CディレクトリとTCINSTの詳細については付録Dを参照してください。

コンフィギュレーションファイルとTCINSTの関係

TCINSTを使えば，Turbo Cのメニューのほとんどすべての項目を設定し，その設定内容を直接TC.EXEの中に保存することができます。このようにカスタマイズを行なったTC.EXEを起動した場合，TCコンフィギュレーションファイルが見つからなければ，TCINSTによる設定がデフォルトになります。

ただし，TC.EXEを起動したときにデフォルトディレクトリ(あるいはTurbo Cディレ

クトリ）にコンフィギュレーションファイル TCCONFIG.TC があれば，TCCONFIG.TC における設定が，TCINST によって組み込まれたデフォルトの設定よりも優先されます。

同様に，/c スイッチでコンフィギュレーションファイルを指定して TC.EXE を起動した場合にも，指定したファイルがあれば，その設定が TCINST によるデフォルトに優先します。

コンフィギュレーションファイルの自動セーブ

通常は，Turbo C が現在のコンフィギュレーションファイルをセーブする（ディスクに書き込む）には，**O**ptions/**S**ave options コマンドを実行しなければなりません。しかし，設定を変更したり追加したりした場合に，自動的にコンフィギュレーションファイルがセーブされるようにすることもできます。

これには，**O**ptions/**E**nvironment/**C**onfig auto save を On にしておきます。こうすると，**R**un/**R**un や **F**ile/**O**S shell を選択した場合や，（**F**ile/**Q**uit で）統合環境を終了する際に，コンフィギュレーションファイルが変更されていれば，自動的にディスクに書き込まれます。

コンフィギュレーションファイルをセーブしていない状態で Config auto save を on にすると，Turbo C は自動セーブを行なうファイル名を決定します。このファイル名は，最後にセーブした（あるいはロードした）コンフィギュレーションファイルになります。また，ロード/セーブを一度も行っていない場合には，（カレントディレクトリの）TCCONFIG.TC になります。

ピックリスト，ピックファイルとはなにか？

ピックリストとピックファイルの2つは，編集セッションの状態を保存しておくために連動して働く Turbo C 統合開発環境の特徴的な機能です。ピックリストは，統合環境の中にいるあいだにどんなファイルを編集している（エディタにロードしたことがある）かを記憶しています。ピックファイルは，統合環境を終了したあとで，あるいは状況を変更したあとで，どんなファイルを編集していたかを記憶しています（状況の変更とは，新たにコンフィギュレーションファイルをロードする，あるいは新たにピックファイル名を定義することを言います）。

ピックリスト

ピックリストは，File/Pickを選択するか，あるいはそのホットキー**GRPH-F3**を押せば呼び出すことができます。File/Pickを選ぶと，それまでにエディタにロードされた最後から8つまでのファイルのリスト（ピックリスト）が表示されます。リストの一番上のファイルは現在エディタの中にあるものです。ピックリストに2つ以上のファイル名がある場合は，リストの2番目のファイル名がハイライト表示されます。このファイルは現在のファイルの前にエディタにロードされていたものです。

ピックリストのファイルをエディタにロードするには，矢印キーでハイライトバーをロードしたいファイル名に動かして**リターン**を押します。こうすると，Turbo Cは指定されたファイルをエディタにロードし，エディタはカーソルをそのファイルの前回の編集終了時の位置に動かします。さらに，このファイルのブロックやプレースマーカも前回の編集終了時と同じ状態に復元されます。

ピックリストは，プログラムの開発中にいくつかのファイルの間を行ったり来たりする際に便利な機能です。**GRPH-F3**，**リターン**の順でキーを押せば，2つのファイルの間を往復することができます（エディタの中にいるときには，**GRPH-F6**を押せばキー1つで同じことが行なえます）。

編集したいファイルがピックリストにない場合には，-- load file --（ピックリストの一番下）を選択します。すると，Load File Nameボックスが現われて，ロードするファイル名を入力することができます（MS-DOS形式のワイルドカードも使えます）。**F3**を押せば自動的にFile/Loadを選択することもできます。

ピックファイル

ピックファイルには，ピックリストの内容も含めて，ファイル関連の情報が保存されます。ピックリスト内の各エントリ（ファイル）ごとに，ファイル名，カーソル位置，ブロックマーカの位置，プレースマーカの位置が記録されます。

各ファイルに関する情報に加えて，ピックファイルにはエディタの終了時の状態についての情報も含まれます。これには，最後の探索/置換文字列や探索オプションなどがあります。

ピックファイルを作成するには，ピックファイル名を定義しなければなりません。これは，Options/Directories/Pick file nameを選択してファイル名を入力します。ピックフ

ァイル名を定義されていれば，統合環境の終了時には必らず Turbo C はディスク上のピックファイルを更新します。

ピックファイルはいつ，どうやって得られるか？

ピックファイルに関する情報を探すことのできるメニュー項目としては，**O**ptions/**D**irectories/**P**ick file name と **O**ptions/**D**irectories/Current pick file の2つがあります。

Q：ピックファイルがすでにあるかどうかはどうすればわかるか?

A：**O**/**D**/Current pick file が空白でなければ（ファイル名が表示されていれば）すでにピックファイルがあることになります。

Q：**O**/**D**/Current file name にどうしてファイル名が現われるのか？

A：**O**/**D**/**P**ick file name にはっきりとファイル名が示されていればその名前が表示されます。あるいは（**O**/**D**/**P**ick file name が空白の場合には）デフォルトのピックファイルがロードされているためです。

Q：**O**/**D**/**P**ick file name にファイル名が示されている場合，そのファイル名はどこから得られたのか？

A：**P**ick file name のファイル名は次の場合に得られます。

- ■現在のセッションでユーザによって入力された。
- ■前回のセッションで入力された内容がコンフィギュレーションファイルにセーブされ，そのコンフィギュレーションファイルが現在のセッションで使われている。
- ■ TCINST によってそのファイル名が組み込まれている。

Q：**O**/**D**/**P**ick file name が空白であるのに，**O**/**D**/Current pick file にはファイル名が示されているとすると，そのデフォルトのピックファイルはどうやってロードされたのか?

A：デフォルトのピックファイル TCPICK.TC がカレントディレクトリにあり，Turbo C が起動時にそれを自動的にロードしたためです。

ピックファイルをセーブすると，統合環境はそのフルパス名を記憶します。この情報は **O**/**D**/Current pick file に表示されます。

Turbo C はピックファイルをいつセーブするか?

統合環境を終了するときには，**O/D/**Current pick file に表示されているピックファイルは必らずセーブされます。また，**O/D/P**ick file name によってピックファイルが変更された場合にも，元のピックファイルはセーブされます（別のコンフィギュレーションファイルのロードによって，ピックファイルが間接的に変更された場合も同様です）。

Current pick file が空白の場合には，統合環境終了時にもピックファイルはセーブされません。

第IV部:追加された機能とエディタコマンド

統合環境で使用可能な Turbo C の便利な機能のいくつかは，メニューシステムの中からはアクセスできません。この節では，ソースコードの編集中に威力を発揮するこうした機能について説明します。

タブについて

エディタのタブモードがオフのときに ***TAB*** キーを押すと，カーソルは次の“ソフト”タブ位置まで移動し，その間には必要な数の空白文字（0x20）が挿入されます。ソフトタブ位置は，カーソルがある行の1つ上の行の各語の頭文字と同じ桁にセットされています。

タブモードがオンのときに ***TAB*** キーを押すと，カーソルは次の“ハード”タブ位置に移動し，必要な数の空白文字が挿入されます（最適フィルモードがオンのときには，文字数が最小となるようにタブ文字（0x09）と空白文字が挿入されます。最適フィルモードについてはすぐあとで説明します）。ハードタブ位置は，デフォルトでは8桁おきにセットされており，Options/Environment メニューの中の Tab size オプションで，2～16の範囲で設定することができます。

タブモードがオンのときには，エディタのステータス行に Tab と表示されています。タブモードのオン/オフは，***CTRL-O T*** によって切り換えることができます。

ソーステキスト中のマークしたブロックを，***CTRL-K W*** コマンドによってエディタからファイル（または PRN）に送るときに，エディタはすべてのタブ文字を“ハードタブ”として扱い，そのまま書き出します。これは一般に8桁ごとのタブ位置をうみだすことになります（プリンタの設定によっては無視されることもあります）。ブロックをプリンタへ送るときに，***CTRL-K W*** のかわりに ***CTRL-K P*** を使うと，エディタはタブ文字を“ソフトタブ”として扱い，それを適切な数の空白文字に置き換えてからプリントします（このときのタブ位置は，O/E/Tab size で指定されている桁数にしたがって計算されます）。

自動インデント,逆インデント,最適フィル

Autoindent（自動インデント）はエディタの機能の1つで，（ハード）**リターン**を入力すると，カーソルが，前の空白でない行の最初の非空白文字と同じカラムに位置づけされることです。

TCINST（Turbo C カスタマイズプログラム）を初めて実行したときには，自動インデントモードは自動的に On になっています。これは，TCINST の **O**ptions/**E**nvironment/**O**ptions for editor/**A**utoindent トグルのデフォルトが On だからです。

Edit ウインドウの中では，***CTRL-O I*** あるいは ***CTRL-Q I*** を押す(つまり，***CTRL*** キーを押したまま ***O*** または ***Q*** を押し，続いて ***I*** を押す)と，自動インデントモードのオン/オフを切り換えることができます。

逆インデントはカーソルのインデント（段下げ）を解除する機能です。つまり，カーソルを左に移動させてその前のインデントレベルに揃えることです。

```
a = 3;
i = 1;
while (i <= 25)
      {
      product = a * i;
      printf("%d.  %d", i, product);
         ++i;
      }
```

←自動インデントによってこの桁にカーソルが来る

←BS を1回押すとインデントが解除されてこの桁に戻る

図 2.24　逆インデントの働き

逆インデントするには，その行の最初の非空白文字，あるいは空白行にカーソルを位置づけて ***BS*** を押します。するとカーソルはその前のインデントレベルに移動します。この場合，***BS*** によって複数桁カーソルが戻されることになります。

最適フィルモードは，同時にタブモードもOnになっていないと効果がありません。この2つのモードがOnになっているとき，自動インデントあるいはアンインデントされている行の先頭は，タブ文字と空白文字によって最適化されて埋められます。これによって，その行の文字数が最小になります。

最適フィルモードのオン/オフは，***CTRL-O F***によって切り換えることができます。

例

■Options/Environment/Tab sizeが8にセットされており（タブ位置は，1,9,17,25,・・・），自動インデントモード，タブモード，挿入モードはすべてオンの状態で，カーソルは27桁目から始まる行の最後にあるとします。

・**リターン**を押して新しく1行を挿入すると，エディタはカーソルをその新しい行の27桁目に位置づけます。

・カーソルを動かさずに，その新しい行に1文字をタイプすると，エディタはその文字の前までを3つのタブ文字（24桁まで）と2つの空白文字（26桁まで）で埋め，トータル5文字が詰め文字として挿入されます。

■上と同じ操作を，Options/Environment/Tab sizeが5にセットされた状態（タブ位置は，1,6,11,16,21,26,・・・）で行なうと，エディタはその文字の前に5つのタブ（25桁まで）と空白1つを挿入します。

■Tab sizeが6にセットされた状態（タブ位置は，1,7,13,19,25,・・・）で，カーソルを18桁目に動かしてから1文字をタイプすると，エディタはその文字の前に2つのタブ（12桁まで）と5つの空白（17桁まで）を挿入します。

ペアマッチ

たくさんの関数，カッコに囲まれた式，ネストしたコメント，また区切りのペアに挟まれた他の構造で満たされているソースファイルをデバッグすることを考えてみてください。実際そのファイルは，次のようなペアを使って組み立てられているはずです。

■中カッコ　　{ }
■不等号記号　〈 〉
■丸カッコ　　( )
■大カッコ　　[ ]
■コメント記号　/* */
■二重引用符　" "
■単引用符　　' '

特定の構造の始まりと終わりを示す区切り記号のペアの対応を調べるのは，ややトリッキィです。いま，下位の式が何重にもネストした複雑な式があって，そのすべてのカッコが正しく対応しているかどうかを調べたいとします。あるいは，数画面分に渡る関数の先頭にいて，その関数の最後にジャンプしたいとします。Turbo C エディタのペアマッチコマンドを使えば，こうした問題を指先で解決することができます。これには以下のような操作を行ないます。

1. まず，問題となる区切り記号（たとえば数画面分の関数の始まりを示す中カッコ）にカーソルを置いてください。
2. 選択した区切り記号に対応するもう一方の区切り記号の位置を見つけるために，***CTRL-Q*** [を押してください(上の例で言えば，もう一方の区切りは関数の終わりを示す中カッコです)。
3. エディタは，選択した区切り記号に対応するもう一方の位置に，即座にカーソルを移動させます。カーソルが意図したとおりの区切り文字に移動すれば，その間のコードには，同じ型でマッチしていない区切り文字はないということがわかります。カーソルが意図とは違う区切り文字に位置したとすれば，やっかいなトラブルに巻き込まれており，ソースを調べて問題点を洗い出す必要のあることがわかります。

ペアマッチに関するいくつかの詳細

Turbo Cエディタの"ペアマッチ"コマンドの基本について述べました。次に，このコマンドを使って，何ができるのか，また何ができないのかを少し詳しく説明します。また憶えておいてほしい注意点についてもお話しします。この節では，次のようなポイントをカバーしています。

- ■実際にはペアマッチコマンドは2つあります。1つは前向き（ファイルの最後に向かって）の探索を行なう ***CTRL-Q*** [，もう1つは後ろ向き（ファイルの先頭に向かって）の探索を行なう ***CTRL-Q***]です。
- ■エディタがコメントの区切り（/＊ ＊/）を探す方法は，他の区切り記号とはやや異なります。
- ■対応する区切り記号が見つからない場合は，カーソルは移動しません。

方向つき/方向なしのペアマッチ

方向が決まっているペアとそうでないペアがあるため，ペアマッチコマンドも2つ必要になります。

たとえば，左中カッコ"{"または左大カッコ"["に対応するカッコを見つけたいとします。選択したカッコより前に対応するカッコがないことは明らかなので，エディタは対応するカッコを前向きに探索します。左中カッコと左大カッコは方向が決まっているので，対応するカッコをどちら向きに探せばよいかは明らかになるため，ペアマッチコマンドはどちらを使っても探索することができます。どちらのコマンドを与えても，エディタは正しい方向で探索を行ないます。

同様に，右中カッコ"}"や右丸カッコ")"の対応を調べるときには，選択したカッコより後ろに対応するカッコがあるはずはないので，エディタは自動的にファイルの先頭に向かって探索を行ないます。これらの区切り文字も方向が決まっており，どちらのペアマッチコマンドを使っても，エディタは正しい方向で対応を調べます。

しかしながら，二重引用符（"）と単引用符（'）の対応を調べるときには，どちら向きに探索すればよいかを自動的に決めることはできません。この場合には，正しいペアマッチコマンドを与えて，探索の方向を指示してやる必要があります。***CTRL-Q*** [を与えると，エディタは前向きに対応するペアを探索し，***CTRL-Q***]を与えると，後ろ向きに探索します。

次の表は，各区切り記号のペアについて，方向が決まっているかどうか，ネストすることができるかどうかをまとめたものです（ネスト可能な区切り記号についてはこのあとで

説明します)。

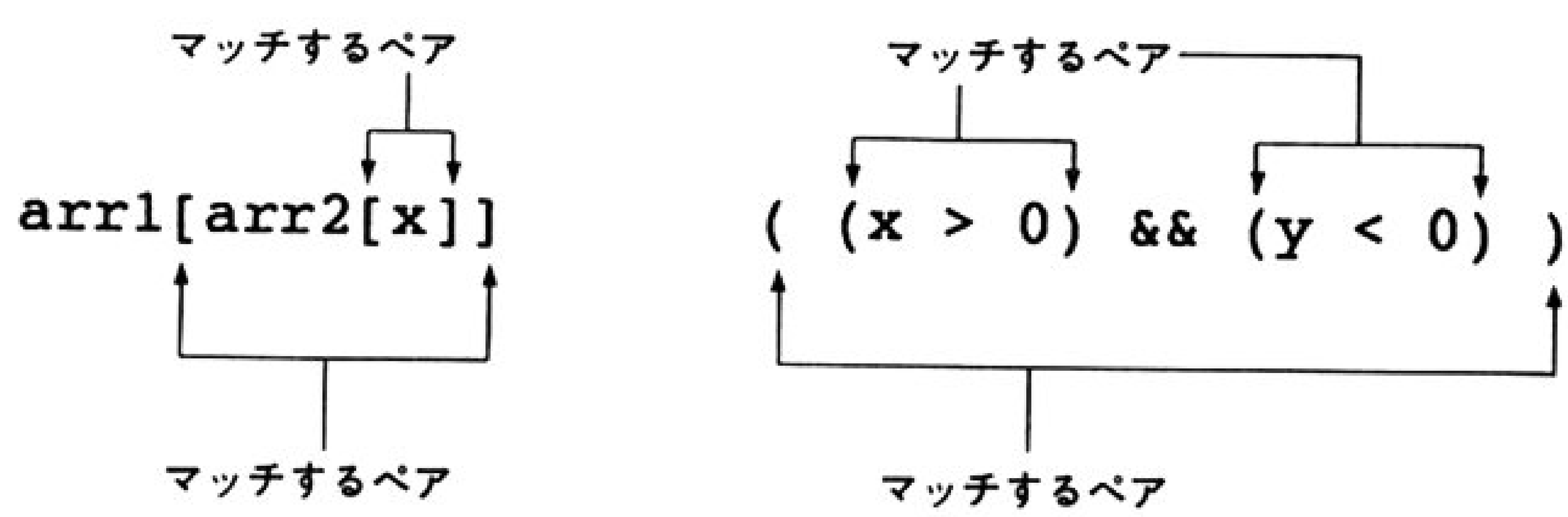
区切り記号	方向	ネスト
{ }	あり	可
( )	あり	可
[ ]	あり	可
< >	あり	可
/* */	あり	場合による
" "	なし	不可
' '	なし	不可

ネスト可能な区切り記号

ネスト可能とはどういう意味でしょうか?これは，方向つきの区切り記号のペアを探索するときに，エディタが探索中に入ったり出たりする区切り記号のレベルを数えていくことを言います。

例を示せばすぐにわかるでしょう。

図 5.25 大カッコまたは丸カッコのマッチ



コメント区切り記号の探索

コメントの区切りは2文字で構成されているために，ペアマッチの探索を行なうときの文字の選択には注意が必要です。どちら向きの場合でも，エディタは2文字のうちの最初の文

字（/*のスラッシュ"/", */のアスタリスク"*"）しか認識しません。どちらの場合でも，2文字目にカーソルが置かれていると，エディタには何を探せばよいのかがわからないので，まったく何もしません。

また，前の表に示したとおり，コメントの区切り記号はネスト可能なときとそうでないときがあります("場合による")。これは，奇怪だとか決めることができないとかいうことではなく，複数の条件に依存する事項です。ANSI 標準案に準拠した C のプログラムでは，コメントをネストすることはできませんが，Turbo C では **O**/**C**/**S**/**N**eted comments トグルの On/Off によってネストを許すことも禁止することもできます。この機能は，コメントの区切り記号のペアマッチを調べるときに，それがネスト可能かどうかにも影響を与えます。

■ O/C/S/Nested comments トグルが On の場合には，エディタはコメントの区切り記号をネスト可能として扱い，ペアマッチを探索するときにそのレベルを数えます。

■ O/C/S/Nested comments トグルが Off の場合は，エディタはコメントの区切り記号をネスト不可として扱います。つまり，/＊が選択されているときには，それより後ろにある最初の＊/が対応するペアとして見つけ出されます。

この違いを示す例をいくつかあげましょう。

図 5.26　Nested comments トグルが On のときの前向き探索

```
/* /* /* /* Here are some nested comments */ */ */ */
```

選択したマッチレベル　　発見されたマッチレベル

注意：O/C/S/Nested comments トグルが On のときに，上の図の「発見されたマッチレベル」の位置で後ろ向き探索を行なうと，見つけられるのは，上の図で「選択したマッチレベル」の位置になります。

図 5.27　Nested comments トグルが Off のときの前向き探索

```
/* /* /* /* Here are some nested comments */ */ */ */
```

選択したマッチレベル　　発見されたマッチレベル

図 5.28　Nested comments トグルが Off のときの後ろ向き探索

```
/* /* /* /* Here are some nested comments */ */ */ */
```

発見されたマッチレベル　　選択したマッチレベル

注意：コメント，二重引用符のペア，単引用符のペア，あるいは条件コンパイルセクションの中にある対応するペアのない区切り記号が，マッチしているペアの間に挟まれていると，これはペアの探索に影響を与えてしまいます。たとえば次のような場合です。

```
main()
{
    ...
    /* Next } is the end of main */
}
```

ここで，**main** 関数の始まりの中カッコを選択して探索を行なうと，見つけ出されるのはコメント内の右中カッコになります。また，**main** 関数の終わりの右中カッコを選択して探索すると，対応する左中カッコは見つかりません（コメント内の右中カッコによってネストのレベルが1つ下がってしまうためです）。

編集キーの割り付け

Turbo C エディタは，なんらかのキーあるいはキーの組み合わせに割り付けられた数多くの編集機能を提供しています。これらの詳細は付録 A で解説されています。

TCINST は，Turbo C 統合環境（TC.EXE）のカスタマイズプログラムです。この中のメニュー項目によって，Turbo C エディタの編集機能を別のキーに割り付けることができます。これはキーの再結合とよばれます。

TCINST の操作の詳細は付録 D で説明されているので，ここでは編集キーの割り付けの基本事項についてだけを述べることにします。Turbo C エディタの編集コマンドを変更するときは，次の一般的な手続きにしたがってください。

1. TCINST.EXE をロードします（DOS のプロンプトから tcinst とタイプして**リターン**を押します）。メインメニュー（Installation Menu）から **Editor commands** を選択してください。Install Editor 画面が現われ，3列のデータが表示されます。

   ■最初（左）の列は，使用できる編集機能の説明です。
   ■2番目の列は第1キーストローク（その編集機能を呼び出すキー）です。
   ■3番目の列は第2キーストローク（同じ編集機能を呼び出すもう1つのキー）です。

2. Install Editor 画面の一番下には，第1および第2キーストロークの設定を変更する際に使用するキーがまとめられています。上下左右の矢印キーを使うと，ハイライトバーを望みのコマンドの第1または第2キーストロークに動かすことができます。

そこで**リターン**キーを押すと，キーストローク編集モードに入ります。

3. 矢印キーを使って，ハイライトバーを変更したいコマンドの第1または第2キーストロークに動かしてください。

4. **リターン**を押して編集モードに入ります。現在定義されているキーストロークを示すポップアップウィンドウが現われます。

5. ***BS*** を押してウィンドウの中のキーストロークを右から左に消していくか，F3を押してウィンドウ内の定義をそっくりクリアします。

6. キーストロークの組み合わせは，WordStar-like，Ignore case，Verbatim の3つの方式から選ぶことができます。***F4*** を押すと3つの方式が順に切り換わり，画面の一番下のハイライトによって現在の方式が示されます。3つの方式の詳細については付録Dを参照してください。

7. その編集機能の新しいキーストロークの定義を入力します(最大6キーストロークまでです)。定義するキーを間違えたときは，***BS*** を押して消すことができます。新しい割り付けをキャンセルしたいときは，***F2*** を押すと変更前の元の割り付けが復元されます。また，途中で ***ESC*** を押せば，元の割り付けを復元してキーストローク編集モードから出ることができます。

8. 新しいキーストロークの入力が済んだら，**リターン**を押してその定義を書き込みます。

9. 必要なキーの割り付けが完了したら，***ESC*** を押して Install Editor 画面から出てメインメニューに戻ります。Quit/save を選択すると，変更内容を TC.EXE にセーブするかどうかを聞いてくるので，変更内容に問題がなければ ***Y*** を押します。***N*** を押せば，すべての変更をキャンセルすることができます。

注意：Turbo C の標準のホットキーを無効にすると（ホットキーと同じキーストロークを他の編集コマンドに割り付けると），エディタの中からはそのホットキーは使えなくなります。

第6章

Turbo C でのプログラミング

いままでにC言語でプログラムを書いたことがあるでしょうか。Cは覚えるのがどんなにむずかしい言語かという話をいろいろ耳にしたかもしれませんが，これはほとんどナンセンスな話です。たしかにCのプログラマの中には，読みにくくデバッグしにくい不明瞭なプログラムを書くことに，むしろ喜びを見いだしている人もいます。しかし，Turbo Cのユーザが，こうした人達のいうとおりにする必要はまったくありません。Cというプログラミング言語の基本的な要素は，本来理解しやすく使いやすいものなのです。

この章では……

この章では，C言語の基本要素について説明し，それらをプログラムの中でどのように使うのかを示しています。次の第7章「**続:Turbo C でのプログラミング**」では，C言語についてよりくわしく説明しており，また第12章「**Turbo C での上級プログラミング**」では，メモリモデルや割り込み，アセンブリ言語でのプログラミング，そのほかやや高度な話題をとりあげています。

もちろん，1つや2つの章で，C言語のプログラミングのすべてについて述べるのは不可能です。C言語のすべてを網羅するような書籍も数多く出版されています。このマニュアルの最後の「**参考書籍**」にC言語に関する書籍の一覧があるので参考にしてください。

この章にしたがって学習を進めていく前に，（もし読んでいなければ）第5章「**Turbo C 統合開発環境**」に目を通して，Turbo Cのメニューとエディタの基本的な使い方を憶えておいてください。また第1章で説明したように，マスターディスクのバックアップコピーをとって，Turbo Cのインストールを済ませておく必要があります。

以上のすべてが完了したら，コンピュータの前に腰掛けて，電源を入れ（まだ入っていなければですが），Turbo Cのプログラミングを学ぶ用意をしてください。

プログラミングの７つの基本要素

ほとんどのプログラムの目的は，問題を解決するということにあります。プログラムは情報やデータを操作することによって問題を解決していきます。プログラミングでしなければならないのは，以下のようなことです。

■情報をプログラムに取り込む
■情報をとっておく場所を確保しておく
■情報を操作する命令を与える
■プログラムから情報を引き出してユーザに示す

また，命令を組織だてて，以下のようなことを行ないます。

■ある特定の条件（または条件の組み合わせ）のもとでのみ実行する
■何度も繰り返し実行する
■いくつかのグループにわけて，プログラム内の別々の場所で実行する

以上がプログラミングの7つの基本要素です。それぞれを一言でいえば，**入力**，**データ型**，**演算**，**出力**，**条件による実行**，**ループ**（繰り返し），**サブルーチン**の7つになります。この7つですべてのプログラムを包括できるわけではありませんが，一般的には，プログラムが共通に持っている要素はこの7つと言えるでしょう。

ほとんどのプログラミング言語はこうした要素をすべて持っていますし，多くの言語は（C も含めて）さらにいろいろな特徴をそなえています。しかしながら，ある言語をすばやく習得したいというときには，この7つの要素がその言語にどのように組み入れられているかを探り出して，そこから組み立てていくのが近道と言えます。ここでは，7つの要素を要約して示しておきます。

入力:キーボードやディスク，あるいはI/Oポートなどから値を読み込むことです。
データ型:数値（整数と実数），テキスト（文字や文字列），アドレス（変数や構造体）などを持つ定数，変数，構造体のことです。
出力:画面やディスク，あるいはI/Oポートなどに情報を書き出すことをいいます。
演算:ある値を他の値に代入したり，値を結び合わせたり（加える，割るなど），比較したり（等しい，等しくないなど）することです。

条件による実行:特定の条件が'真'であれば命令を実行する（'偽'であれば実行しない）ことを言います。

ループ（繰り返し）:決められた回数，あるいは条件が真である間，命令を実行することです。

サブルーチン:名前のついている命令のグループのことで，その名前を指定すればプログラムのどこからでも実行することができます。

それでは，Turbo C でこうした要素をどのように使うのかを見ていきましょう。

出力

最初に出力について述べるのはおかしいと思われるかもしれませんが，情報をまったく出力しないプログラムはあまり使い道がないことは明らかです。出力は通常，いろいろな装置に書き出される情報という形式をとります。装置には，画面（言葉や絵），外部記憶装置（フロッピィやハードディスク），あるいはI/Oポート（シリアルポートやプリンタポート）といったものがあります。

printf 関数

C 言語でもっともよく使われる出力の関数を，すでに何度か使いました。**printf** ルーチンです。**printf** の目的は，画面に情報を書くことです。その書式は，簡単でフレキシブルなものになっています。

```
printf(書式文字列, 項目, 項目, ...);
```

書式文字列

書式文字列は，二重引用符で囲まれた文字列です（"like this"）。**printf** の目的は，その文字列を標準出力（*stdout*）に書くことです。まず **printf** は，書式文字列の中にある**書式コマンド**にしたがって，書式文字列の後ろに並んでいる項目を書式文字列の中に組み入れていきます。第2章に出てきたプログラム SUM.C にあった，次のような **printf** 文を見てみましょう。

```
printf("The sum is %d ¥n",sum);
```

書式文字列の中の%dが**書式指定**と呼ばれる書式コマンドの一種です。書式指定はすべてパーセント記号（%）で始まり，そのあとに（たいていは）英字1文字が続いて，そこに組み入れられるデータの型と書式の種類を指定します。

書式指定1つに対しては，項目が1つ対応していなければなりません。項目のデータ型が，書式指定が示している型と直接対応していない場合には，期待どおりの結果が得られなくなります。項目自体には，変数，定数，式，関数呼び出しを使うことができます。要するに，項目は，対応する書式指定に対して適切な値を持っていれば何でもよいわけです。

上の指定の中で使われている%dは，これに対応する項目として整数が想定されていることを意味しています。よく用いられる書式指定をいくつか示しておきます。

%u	符号なし整数
%ld	倍長整数
%p	ポインタ値
%f	浮動小数点数
%e	指数表記の浮動小数点数
%c	文字
%s	文字列
%x または %X	16進表記の整数

%と英字の間に数字を置くと，そのフィールドの幅を示すことになります。たとえば，4桁分の整数のフィールドを設定するには，%4dと書くことになります。この指定では，値は右寄せで出力され(数値の桁数が少なければ先頭に空白が入って)，フィールド全体の幅が4桁分になります。

パーセント記号そのものを書きたい場合は%%とします。

文字列内の¥nは書式指定ではありません。これは(歴史的な理由から)**エスケープシーケンス**として知られるもので，文字列の中に挿入される特殊な文字を表わします。この場合では，¥nは復帰改行文字を挿入するので，文字列が書き出されると，カーソルは次の行の先頭に移動します。

エスケープシーケンスについては第11章に完全なリストがありますが，よく使われるものをいくつかあげておきます。

¥f	改ページまたは画面クリア
¥t	タブ
¥b	バックスペース
¥x*hhh*	ASCIIコード *hhh* で表わされる文字，*hhh* は1～3桁の16進数

円記号（¥）を出力したい場合は，¥¥と2つ続けます。**printf** の働きについてもっと詳しく知りたい方は，リファレンスガイド第2章の **printf** の解説を参照してください。

他の出力関数：puts と putchar

他にも興味深い出力関数が2つあります。**puts** と **putchar** です。

puts 関数は，文字列を画面に書いてから復帰改行文字をつけます。たとえば，HELLO.Cを次のように書きなおすことができます。

```
#include <stdio.h>

main()
{
    puts("Hello, world");
}
```

文字列の最後の¥n をはずしていることに注意してください。**puts** が¥n を付け加えるので，必要ないわけです。

putchar 関数は，画面に文字を1つ書くだけで¥n はつけません。putchar(ch) という文は，printf("% c",ch) と同じ働きをします。

puts や **putchar** は，どんな理由で **printf** のかわりに使うことになるのでしょうか。理由の1つとしては，printf を組み込んだルーチンは比較的大きくなってしまうということがあります。数値の出力や特殊な書式のために **printf** を必要とするのでなければ，かわりに **puts** や **putchar** を使うことによって，プログラムをより小さくまた速くすることができます。HELLO.C でも，**puts** を使った場合と **printf** を使った場合とで，.EXE ファイルの大きさを比べると，puts を使ったほうがずっと小さくなります。

データ型

プログラムを書くときにはなんらかの情報をとり扱うことになりますが，その情報のほとんどは，4つの基本的な型に分類することができます。すなわち，**整数**，**浮動小数点数**，**テキスト**，**ポインタ**の4つです。

整数は，数をかぞえるのに使われます（1，5，-21，752など）。

浮動小数点数には，小数（3.14159など）と，べき指数（2.579×10^{24}など）があります。浮動小数点数は，実数（real）と呼ばれることもあります。

テキストは，文字（a，Z，!，3）と文字列（"This is only a test."）から構成されます。

ポインタは，情報そのものではなく，コンピュータのメモリ上でその情報が記憶されている場所を示すアドレスを持っています。

浮動小数点型（float）

C言語は，これらの4つの基本データ型を，さまざまな形式でサポートしています。その中の2つは，いままでのプログラムで使用しました。整数（int）と文字（char）です。では，前回のプログラムを修正して，3番目の型――浮動小数点数(float)を使ってみましょう。

Turbo Cのエディタに入って，プログラムSUM.Cを以下のように変更してください。

```
#include <stdio.h>

main()
{
    int    a,b;
    float  ratio;

    printf("Enter two numbers:  ");
    scanf("%d %d",&a,&b);
    ratio = a / b;
    printf("The ratio is %f ¥n",ratio);
}
```

プログラムの修正がすんだら，メインメニューに入ってFile/Write toコマンドを選択し，RATIO.Cという名前でセーブしてください。それから***CTRL-F9***を押して，プログラムをコンパイルして実行してください。2つの値（たとえば10と3）を入力して，出力された結果（3.000000）に注目してください。

3.333333という答えを期待されたのではないでしょうか。どうして答えがぴったり3なのでしょう。これは，*a*と*b*がともに整数型(int)だからで，したがって*a*/*b*の結果も整数型になるのです。*a*/*b*の値は，*ratio*に代入されるときにfloatに変換されますが，変換は除算の前ではなく後で行なわれます。

もう一度エディタに戻って，*a*と*b*の型をfloatに変更してください。また，**scanf**文の中の書式文字列"%d%d"も，"%f%f"になおしてください。変更したコードをセーブし（***F2***を押す），コンパイル・実行してみてください。今度は，結果は期待どおり3.333333になります。

floatを大きくしたバージョンとして，doubleおよびlong doubleという2つの浮動小数点型があります。呼び名（ダブル）からもわかるように，double型の変数は，float型の2倍の大きさを持っており，またlong double型はこれよりさらに大きいものです。サイズが大きいということは，より多くの有効桁数を持ち，また指数の範囲もより広いというこ

とを意味します。Turbo C で，これらの型のメモリ上のサイズと，値のとり得る範囲がどのようになっているかは第11章で述べられています。

3つの整数型

int 型に加えて，C は short int と long int という2つの整数型をサポートしており，普通 short および long と略称されています。short，int，long の実際のサイズは処理系に依存します。C が保証しているのは，short 型の変数は long 型の変数より大きくてはいけない（つまり，より多くのバイトを占めてはいけない）ということだけです。Turbo C では，short は16ビット，int は16ビット，long は32ビットというサイズになっています。

符号なし整数（unsigned）

C 言語では，いくつかの型（char，short，int，long）は，符号なし（unsigned）であると宣言することができます。符号なしというのは，その型は負の値は持たない，つまり0または正の値のみを持つことを意味しています。

符号なしの変数は，符号つきの変数より大きな値を持つことができます。Turbo C では，int 型は-32768～32767の値を持つことができるのに対し，unsigned int の値の範囲は0～65535です。どちらも同じだけのメモリ空間（16ビット）を占めていますが，その空間の使い方が異なっているわけです。詳細については第11章を参照してください。

文字列の定義

C は，文字列という独立したデータ型はサポートしていませんが，文字列を定義する方法は2つあります。1つは**文字配列**を使うもので，もう1つは**文字ポインタ**を使用するものです。

文字配列を使う

File メニューから Load コマンドを選択して，もう一度 HELLO.C を読み込んでください。今度は，次のように修正します。

```
#include <stdio.h>
#include <string.h>

main()
{
    char  msg[30];

    strcpy(msg,"Hello, world");
    puts(msg);
}
```

msg の後の[30]は，29文字分のメモリ領域を確保するようにコンパイラに指示しています。つまり，char 型変数29個の配列を宣言しているわけです。配列の30番目の要素はヌル文字（¥0）になります。*msg* という変数それ自体は，文字の値を持っているわけではなく，29個の char 型変数の最初のアドレス（メモリ上の場所）を保持しています。

次の文があると，

```
strcpy(msg,"Hello, world");
```

コンパイラは2つのことを行ないます。

- ■"Hello, world"という文字列を作成して，最後にヌル文字（¥0，ASCII コードの0）をつけて，オブジェクトコードのどこかに置きます。
- ■**strcpy** というサブルーチンを呼び出すコードを生成します。**strcpy** は，上で作成した文字列を，*msg* が示すメモリ位置へ1文字ずつ順にコピーしていきます。"Hello, world"という文字列の終わりについているヌル文字をコピーすると，コピーは終了します。

puts(*msg*)を呼び出すと，*msg* の値（*msg* が指している最初の文字のアドレス）が **puts** に渡されます。すると **puts** は，そのアドレスにある文字がヌル文字かどうかを調べます。ヌル文字であれば **puts** は処理を終了します。ヌル文字でなければその文字を出力し，アドレスに1を加えて再びそのアドレスの文字をチェックします。

このように，文字列の操作はヌル文字に依存しているため，C の文字列は“**ヌルで終了する**”（文字の並びの最後にヌル文字がついている）ということで知られています。この方法で文字列を表現しているので，文字列の長さに関する制限はまったくありません。文字列は，充分なメモリがありさえすれば，どんな長さになってもかまわないわけです。

文字ポインタを使う

文字列を定義する2番目の方法は，文字ポインタを使うものです。プログラムを以下のように変更してください。

```
#include <stdio.h>
#include <string.h>

main()
{
    char  *msg;

    msg = "Hello, world";
    puts(msg);
}
```

msg の前にあるアスタリスク（*）は，*msg* が文字を指すポインタであることを示しています。言いかえれば，*msg* は文字のアドレスを持つことができるということになります。しかし，コンパイラは文字をおさめておくメモリ領域を確保しませんし，*msg* を特定の値で初期化することもしません。

次の文があると，コンパイラは2つのことを行ないます。

```
msg = "Hello, world";
```

■（文字配列の場合と同じように）"Hello, world"という文字列を作成して，最後にヌル文字をつけ，それをオブジェクトコードのどこかに置きます。

■この文字列の先頭アドレス（H のアドレス）を *msg* に代入します。

puts(*msg*)という文は前と同様に機能し，ヌル文字に出会うまで文字を出力します。

文字列を定義する2つの方法,配列を使う場合とポインタを使う場合では微妙な違いがありますが，それについては次の章で説明することにします。

識別名

いままで，変数につける名前に関しては，どんな制限があるのかなどということは意識せず，まったく自由に決めてきました。ここで，その制限についてふれておきましょう。

定数，データ型，変数，および関数につける名前は，**識別名**と呼ばれています。これまでに使用した識別名をいくつかあげてみます。

char，int，float	あらかじめ定義されているデータ型
main	プログラムの主関数の名前
name，*a*，*b*，*sum*，*msg*，*ratio*	ユーザ定義の変数名
scanf，**printf**，**puts**	あらかじめ宣言されている関数名

Turbo C には，識別名についての制限がいくつかあります。簡単にまとめると次のようになります。

- ■識別名はすべて，英字（a～z，A～Z）または下線（_）で始まらなければならない。
- ■識別名の2文字目以降は，英字，下線，数字（0～9）を使ってよい。これら以外の文字は使用できない。
- ■識別名は**大文字と小文字を区別する**。つまり，小文字（a～z）と大文字（A～Z）は異なる。たとえば，*indx*，*Indx*，*INDX* はすべて別個の識別名である。
- ■識別名は，先頭から32文字までが意味を持つ（有効である）。

演算

プログラムに（そして変数に）データを与えたら，これをどうしようとするでしょうか。おそらく，適当な演算子を使ってデータになんらかの操作を加えることになるでしょう。C 言語には，非常に多くの演算子が用意されています。

代入演算子

もっとも基本的な演算は，*ratio* = *a*/*b* や *ch* = **getch**() などのような**代入**です。C では，代入は1つの等号（=）で表わされ，等号の右側の**値**が，左側の**変数**に代入されます。

sum = *a* = *b* のように，代入を積み重ねることもできます。この場合には，式の評価

の順序は，右から左になります。したがって，まず *b* が *a* に代入され，次にその値が *sum* に代入されます。その結果，3つの変数はすべて同じ値（*b* の元々の値）を持つことになります。

単項演算子と二項演算子

Cでは，以下のような一般的な二項演算子がサポートされています。

■乗算（*）
■除算（/）
■剰余（%）
■加算（＋）
■減算（-）

Turbo C では，**単項マイナス演算子**（*a* ＋（-*b*））をサポートしており，これは数値の符号を逆転させます。また ANSI 標準化案にしたがって，**単項プラス演算子**（*a* ＋（＋*b*））もサポートしています。

インクリメント（＋＋），デクリメント（--）演算子

Cには，特殊な単項演算子および二項演算子がいくつかあります。もっともよく知られている単項演算子は，**インクリメント**（＋＋）と**デクリメント**（--）です。これらを使うと，演算子を1つつけるだけで，ある値に**1を加える**，またはある値から**1を引く**ことができます。加算または減算を式の途中で行なうことができ，その加算や減算を式の評価の前に行なうのか，評価の後で行なうのかを指定することができます。次の2つの式を考えてみましょう。

```
sum = a + b++;
sum = a + ++b;
```

最初の文は「*a* と *b* を加え，その結果を *sum* に代入し，その後 *b* に1を加える」という意味です。2番目は「*b* に1加えた後，*a* と *b* を加え，その結果を *sum* に代入する」という意味になります。

この2つはとても強力な演算子ですが，使用する前に正しく理解しておかなければなりません。SUM.C を以下のように修正して，実行する前にどんな値が出力されるか考えてみてください。

```
#include <stdio.h>

main()
{
    int    a,b,sum;
    char   *format;

    format = "a = %d    b = %d    sum = %d ¥n";
    a = b = 5;
    sum = a    + b;  printf(format,a,b,sum);
    sum = a++ + b;  printf(format,a,b,sum);
    sum = ++a + b;  printf(format,a,b,sum);
    sum = --a + b;  printf(format,a,b,sum);
    sum = a-- + b;  printf(format,a,b,sum);
    sum = a    + b;  printf(format,a,b,sum);
}
```

ビット演算子

ビットレベルの演算については，Cには以下のような演算子があります。

■左シフト　　<<
■右シフト　　>>
■ AND　　&
■ OR　　|
■ XOR　　^
■ NOT　　~

これらの演算子を使うと，値に対して非常に低レベルの操作が可能になります。これらの演算子の働きを見るために，次のようなプログラムを入力して実行してみてください。

```
#inlude <stdio.h>

main()
{
    int   a,b,c;
    char  *format1,*format2;

    format1 = " %04X %s %04X = %04X¥n";
    format2 = " %c%04X = %04X¥n";
    a = 0x0FF0;  b = 0xFF00;
    c = a << 4;  printf(format1,a,"<<",4,c);
    c = a >> 4;  printf(format1,a,">>",4,c);
    c = a & b;   printf(format1,a,"& ",b,c);
    c = a | b;   printf(format1,a,"| ",b,c);
    c = a ^ b;   printf(format1,a,"^ ",b,c);
    c = ~a;      printf(format2,'~',a,c);
    c = -a;      printf(format2,'-',a,c);
}
```

このプログラムについても，実行する前にどんな出力が行なわれるかを予想してみてください。なお，ここでは出力の位置をきれいにそろえるために，フィールド幅指定子が用いられています。%04X という書式指定は「先頭は0で埋めて，4桁で，16進表記で出力する」ことを意味しています。

演算子の結合（複合代入演算子）

Cでは，複数個の演算子を含む式は，略記できる場合があります。これは，代入演算子（＝）と，いままでに出てきた演算子（単項，二項，インクリメント，デクリメント，およびビット）を結合して使用するものです。

下のような形式の式は，

<変数> = <変数> <演算子> <式>

次のように置き換えることができます。

<変数> <演算子> = <式>

こうした式の例を示しておきましょう。左側の式は，右側の式に置き換えることができます。

```
a = a  + b;     →    a  += b;
a = a  - b;     →    a  -= b;
a = a  * b;     →    a  *= b;
a = a  / b;     →    a  /= b;
a = a  % b;     →    a  %= b;
a = a << b;     →    a <<= b;
a = a >> b;     →    a >>= b;
a = a  & b;     →    a  &= b;
a = a  | b;     →    a  |= b;
a = a  ^ b;     →    a  ^= b;
```

アドレス演算子

Cは，2つの特別なアドレス演算子をサポートしています。**アドレス of 演算子**（&）と，**間接演算子**（*）です。

&演算子は与えられた変数のアドレスを返します。たとえば，int 型の変数 *sum* があると，*&sum* はその変数のアドレス（メモリ上の位置）を表わすことになります。

*演算子はポインタ型変数に対して使用され，そのポインタが指しているもの（オブジェクト）を返します。*msg* が char 型を指すポインタだとすると，* *msg* は *msg* が指している文字ということになります。

次のプログラムを入力して，実行の結果を確認してみてください。

```
#include <stdio.h>

main()
{
    int   sum;
    char  *msg;

    sum = 5 + 3;
    msg = "Hello, there¥n";
    printf(" sum = %d  &sum = %p ¥n", sum,&sum);
    printf("*msg = %c   msg = %p ¥n",*msg, msg);
}
```

最初の **printf** 文は2つの値を出力します。*sum* の値（8）と *sum* のアドレス（コンパイラが割り当てたアドレス）です。2番目の **printf** も2つの値を出力します。*msg* が指し示す文字（H）と，*msg* の値，つまりその文字（H）のアドレス（これもコンパイラが割り当てたもの）です。

注記:単項演算子&は，K&R では名前を付けていませんが，変数のアドレス（address of variable）を与えるところから address-of operator と呼ばれることが多く，このマニュアルではこの意味（～のアドレス）がとりやすいように‘アドレス of 演算子’と訳しています。

入力

Cにはいくつかの入力関数が用意されています。関数によって，ファイルから入力を行なうもの，キーボードから行なうものなどがあります。入力関数の詳細については，リファレンスガイド第2章の**scanf**や**read**などの解説を参照してください。また，このマニュアルの第7章でも入力（および出力）について再び取り上げています。

scanf 関数

対話的な入力を行なう場合，そのほとんどは**scanf**関数を使うことになるでしょう。**scanf**は，**printf**の入力版といえるもので，書式は次のようになっています。

```
scanf(書式文字列, アドレス, アドレス, ...);
```

scanfでは，**printf**と同様な%**英字**という書式指定を使用し，%dは整数，%fは浮動小数点数，%sは文字列に対応します。

しかし，**scanf**と**printf**には重要な違いがあります。書式文字列の後に続く項は，値ではなくアドレスでなければなりません。前に作成したプログラムSUM.Cには，次のような**scanf**の呼び出しがありました。

```
scanf("%d %d",&a,&b);
```

この呼び出しでは，空白で区切った2個の10進数値（整数）の入力を要求しており，最初に入力された値は*a*に結びつけられたアドレスに格納され，2番目は*b*に結びつけられたアドレスに格納されるようになっています。アドレスof演算子（&）を使って，*a*と*b*のアドレスを**scanf**に渡している点に注意してください。

ホワイトスペース

上の**scanf**文の2つの%dの間の空白には，単なる空白文字以上の意味があります。この空白は，入力の際に2つの値の間には，どんなホワイトスペースを入れてもよいということを示しています。**ホワイトスペース**というのは，空白，タブ，改行のすべての組み合せのことです。Cのコンパイラとプログラムは，ほとんどの環境下でホワイトスペースを無視することになっています。

では，空白ではなくてカンマで数字を区切りたいときには，どうすればいいでしょうか。

その場合には次のように書きます。

```
scanf("&d,&d",&a,&b);
```

これで，カンマで区切って値を入力することができます。

scanf にアドレスを渡す

文字列を入力する場合はどうするのでしょうか。以下のプログラムを入力して実行してみてください。

```
#include <stdio.h>

main()
{
    char  name[150];

    printf("What is your name:  ");
    scanf("%s",name);
    printf("Hello, %s¥",name);
}
```

name は文字の配列なので，*name* の値自体は配列のアドレスです。したがって，*name* の前に&演算子をつける必要はありません。単に scanf("% s",name);とすればいいわけです。

ポインタを使う方法（char * name;）ではなく，配列を使う方法（char name[150];）を用いている点に注意してください。どうしてでしょうか。配列を宣言すれば，文字列を格納するメモリ領域は宣言した時点で確保されるのに対し，ポインタを宣言した場合には確保されません。もし，char * name;を使いたいのであれば，* *name* のために実際にメモリを割り当ててやらなければならないのです。

gets と getch を使って入力する

scanf を使って文字列の入力を行なうと，別の問題がもちあがります。もう一度プログラムを走らせて，今度はフルネームとをスペースで区切って入力してみてください。プログラムは，ファーストネームしか表示しないことがわかるでしょう。これは，ファーストネームとファミリィネームの区切りに使った空白が，**scanf** にとっては文字列の終了を意味してしまったからです。

この問題には2つの解決策があります。1つは，プログラムを次のように手直しすることです。

```
#include <stdio.h>

main()
{
    char  first[30],middle[30],last[30];

    printf("What is your name:  ");
    scanf("%s %s %s",first,middle,last);
    printf("Hello, Dr. %s, or should I say %s¥",first,last);
}
```

見てわかるとおり，このプログラムはミドルネームがあることを前提にしており，実際に3つの文字列が入力されるまで，**scanf** は次の動作を行ないません。

それでは，空白などすべてを含めた1つの文字列として，名前全体を取り込みたいとすればどうなるでしょうか。これに答えるのが，2番目の解決策です。

```
#include <stdio.h>

main()
{
    char  name[150];

    printf("What is your name:  ");
    gets(name);
    printf("Hello, %s¥",name);
}
```

gets 関数は，**リターン**キーが押されるまで，タイプされた文字をすべて読み込みます。読み込まれた文字列には**リターン**は含まれず，終わりにヌル文字（¥0）が付加されます。

入力関数の最後に **getch** 関数を紹介しましょう。**getch** は，キーボードから1文字だけを読み込みます。タイプした文字は，**scanf** や **gets** とは違って画面にはエコーバックされません。また，文字型の変数を引数として指定するのではないことを憶えておいてください。**getch** は char 型の関数であり，**getch** の返す値が直接変数に代入されます。

注記:**getch** は，実際には int 型として宣言されています(int getch(void);)。ただ，**getch** は一般的に1文字の入力に使用される関数であり，また char と int は同じものとして扱われる場合が多いので，ここでは混乱を避けるためにこのような記述になっています。

条件文

まだふれていない2種類の演算子――関係演算子，論理演算子があり，また式についても，もう少しこみいった事柄がありますが，これらは，この節で説明する条件文のためにとっておいたのです。

関係演算子

関係演算子は2つの値を比較して，その比較が真か偽かに基づいて結果を与えることができます。比較の結果が偽であれば0を返し，真であれば1を返します。C で使用できる関係演算子には，以下のものがあります。

＞	より大きい
＞＝	より大きいか，等しい
＜	より小さい
＜＝	より小さいか，等しい
＝＝	等しい
!＝	等しくない

どうして，なにかが真であるか偽であるかを問題にするのでしょうか。RATIO.C をロードして実行し，2番目の値に0を入れたらどうなるか試してみてください。プログラムは，Devide by zero というエラーメッセージを出して停止します。

それでは，プログラムを次のように変更して，もう一度実行してみてください。

```
#include <stdio.h>

main()
{
    float  a,b,ratio;

    printf("Enter two numbers:  ");
    scanf("%f %f",&a,&b);
    if (b == 0.0)
        printf("The ratio is undefined¥n");
    else {
        ratio = a / b;
        printf("The ratio is %f ¥n",ratio);
    }
}
```

scanf の次の2行は if 文として知られています。これは「もし式（b == 0.0）の値が真であれば次の **printf** を実行し，式の値が偽であったら *a/b* を *ratio* に代入して **printf** を実行する」という意味になります。

今度は，2番目の値に0を入れると，プログラムは次のメッセージをユーザ画面に表示して Turbo C に戻ります。

```
The ratio is undefined
```

2番目の値が0でなければ，プログラムは *a* と *b* の比を計算して画面に表示します。

論理演算子

論理演算子には，論理積 AND（&&），論理和 OR（||），否定 NOT（!）の3つがあります。前に説明したビット演算子（&, |, ~）と混同しないようにしてください。論理演算子は，論理値（真および偽）の演算を行ない，関係式の結合を可能にします。

論理演算子は，ビット演算子とは次のような違いがあります。

■論理演算子は常に0（偽）・または1（真）という結果を返しますが，ビット演算子はビット対ビットの演算を行ないます。

■&&と||の2つの論理演算子は，式の評価においてショートサーキット（短絡回路）をとる場合があります。たとえば，***exp1* && *exp2*** という式があるとします。***exp1*** が偽であれば，式全体は必らず偽になるはずなので，***exp2*** の評価は行ないません。同じように，***exp1* || *exp2*** という式では，***exp1*** が真であったら ***exp2*** の評価は行ないません。

さらに式について

ループに歩を進める前に，式についてもう少しお話ししておくことがあります。

(b == 0)や(a <= q * r)といった式は非常に簡単なものですが，C 言語では，式はより複雑になってくることがあります。

代入文

カッコ( )で囲まれた代入文はすべて，代入された値と同じ値を持つ式になります。たとえば(sum = 5 + 3)という式はの値は8になります。したがって((sum = 5+3) <= 10)という式は，(8<=10なので）常に真の値を返すことになります。

次の例はちょっと変わったものです。

```
if ((ch=getch()) == 'q')
    puts("Quitting, huh?¥n");
else
    puts("Good move; we'll have another go at it¥n");
```

これが何をするのかおわかりでしょうか。プログラムは，((ch=getch())== 'q')という式を評価すると，文字が入力されるまで停止します。入力があったらその文字を *ch* に代入し，*q* という文字と比較します。*q* と等しければ"Quitting, huh?"というメッセージを画面に表示し，*q* 以外の文字であれば別のメッセージ（"Good move; ・・・"）を表示します。

カンマ演算子（逐次演算子）

カンマ演算子（,）を使うと，カッコ( )の中に複数の式を入れることができます。カッコ内の式は左から右へ順に評価され，式全体の値は最後に評価された式の値になります。たとえば，*oldch* と *ch* がともに char 型の変数だとすると，次の式は，

```
(oldch = ch, ch = getch())
```

まず *ch* を *oldch* に代入し，次にキーボードから文字を取り込んで *ch* に代入します。そして式全体の値は *ch* の値（*ch* に代入された値）になります。

下の例を考えてみてください。

```
ch = 'a';
if ((oldch = ch, ch = 'b') == 'a')
    puts("aye");
else
    puts("bee");
```

if...else 文

前の例の if 文をもう一度見てください。if 文の一般的な形式は次のようになります。

```
if (値)
      文1;
else
      文2;
```

最初2行（if（値）文1;）を if 節，後半の2行（else 文2;）を else 節と呼びます。ここで

の**値**とは，最終的に整数値になる（あるいは変換できる）あらゆる式を指します。もし**値**が0以外（真）であれば**文1**が実行され，0（偽）であれば**文2**が実行されます。

if...else 文については，一般的なこととして2つの重要な点を説明しておかなければなりません。

まず第1は，else **文2**；の部分（else 節）は省略してもよいということです。言いかえると，if 文としてどうしても必要なのは以下の部分（**if 節**）ということになります。

```
if (値)
      文1;
```

文1は**値**が0でない場合にのみ実行されます。**値**が0であったら，プログラムは**文1**を飛び越してその先に進みます。

第2は，特定の式が真（または偽）のときに，複数の文を実行したい場合にはどうすればよいか，という問題です。「**複合文**を用いる」のがその解答です。複合文は，以下のものから構成されます。

■左中カッコ（{）

■それぞれセミコロン（;）で終わる複数個の文

■右中カッコ（}）

RATIO プログラムの例では，if 節には1個の文を使い，

```
if (b == 0.0)
    printf("The ratio is undefined¥n");
```

else 節では複合文を使っています。

```
else {
    ratio = a / b;
    printf("The ratio is %f ¥n",ratio);
}
```

プログラム全体（**main** 関数）もまた複合文であることに気づいた方もいるでしょう。

ループ

条件によって実行したい文（または文のグループ）があるように，繰り返して実行したい文もあるでしょう。この種の文の構造は**ループ**と呼ばれています。

Cには，基本的には3種類のループがあります(その内の2つは，もう1つの特殊な場合ですが)。while ループ，for ループ，do...while ループの3つです。では，順にこの3つを見ていきましょう。

while ループ

while ループはもっとも一般的なループで，他の2つのかわりに用いることもできます。言いかえれば，while ループさえあればどんな要求にも応じられるわけで，他の2つは，あればより便利というものです。それでは，以下のプログラムを入力して実行してみてください。

```
#include <stdio.h>

main()
{
    int  len;

    len = 0;
    puts("Type in a sentence, then press <RETURN>");
    while (getchar() != '¥n')
        len++;
    printf("¥nYour sentence was %d characters long¥n",len);
}
```

このプログラムは，文字列が入力されるときに，**リターン**キー（¥n）が押されるまでにキーが何回押されたかを数え，タイプされた文字数を（**リターン**は含めないで）表示します。

while 文の形式は次のようになります。

```
while (式)
    文;
```

式は最終的に0または0でない値になるもの，また文は1個の文または複合文です。

while ループは，式の値を評価します。式の値が真であれば文を実行し，そのあと再び式の値を調べます。式の値が真でなければ while ループは終了し，プログラムは先に進みま

す。

while ループのもう1つの例として，HELLO.C をもとにして次のプログラムを作ってみましょう。

```
#include <stdio.h>

main()
{
    char  *msg;
    int   indx;

    msg = "Hello, world";
    indx = 1;
    while (indx <= 10) {
        printf("time #%2d:  %s¥n",indx,msg);
        indx++;
    }
}
```

これをコンパイルして実行すると，結果は次のようになります。

```
time # 1:  Hello, world
time # 2:  Hello, world
time # 3:  Hello, world
  ............
time #10:  Hello, world
```

printf 文はちょうど10回実行され，*indx* は10回の実行の間に1から10まで1ずつ増えていきます。

この while 文をよく考えてみると，ループを少し縮められることに気がつきます。

```
indx = 0;
while (indx++ < 10)
    printf("time #%2d:  %s¥n",indx,msg);
```

どうしてこの while ループが，前のものとまったく同じように機能するのか，理解できるまで考えてみてください。それから，for ループに進むことにしましょう。

for ループ

for ループは，C を含めて主なプログラミング言語のほとんどに用意されています。しかしながら，C の for ループは，見ればわかりますが非常にフレキシブルであり，またたいへん強力です。

基本的な考え方は「変数の値がある範囲内で順に変わっていく間に，決められた回数だけ一連の文を実行する」ということになります（この変数はインデックス変数と呼ばれます）。

先程の while ループを使ったプログラムを，以下のように変更してみてください。

```
#include <stdio.h>

main()
{
    char  *msg;
    int   indx;

    msg = "Hello, world";
    for (indx = 1; indx <= 10; indx++)
        printf("time #%2d:  %s¥n",indx,msg);
}
```

実行すればわかるとおり，このプログラムは while ループの例と同じ動作をし，また厳密には while ループの最初の例とまったく同等なものです。for ループの一般的な形式は下のようになります。

```
for (式1; 式2; 式3)
    文;
```

while 文と同様，for 文も1つの文しか実行できませんが，この文はもちろん複合文（{...}）であってもかまいません。

for という語の後にくるカッコの中身に注目してみましょう。セミコロン(;)で区切られた3つの部分があります。

■**式1**は，通常インデックス変数への代入を行ないます。

■**式2**は，ループ（繰り返し）を続けるかどうかのテストをします。

■**式3**は，通常インデックス変数の値を変更します。

一般的な for ループは，以下のような while ループと同じ働きをします。

```
式1;
while (式2)  {
    文;
    式3;
}
```

3つの式のうちのどれか，あるいは全部を省略することができますが，2つのセミコロンを取り除くことはできません。なお，**式2**を取り去ってしまうと，1という値（真）であるとみなされるので，ループは決して終了しなくなってしまいます（これを**無限ループ**と呼びます）。

一方，カンマ演算子を使えば，各々の式として複数の式を書くことができます。HELLO.C を次のように変更してみてください。

```
#include <stdio.h>

main()
{
    char  *msg;
    int   up,down;

    msg = "Hello, world";
    for (up = 1, down = 9; up <= 10; up++, down--)
        printf("%s: %2d down, %2d to go¥n",msg,up,down);
}
```

このforループの**式1**と**式3**はどちらも2つの式からなっており，それぞれ変数*up*と*down*の初期化と変更を行なっている点に注意してください。これらの3つの式はいくらでも複雑にすることができます（for文の3つの式にプログラムのほとんどを詰め込んで，あとはループとして実行される文がほんの少しあるだけというプログラムを書く，Cのハッカー達の伝説を耳にした方もいるかもしれません）。

do...whileループ

ループの最後は，do...whileです。RATIO.Cを以下のように手直ししてください。

```
#include <stdio.h>

main()
{
    float  a,b,ratio;
    char   ch;

    do {
        printf("¥nEnter two numbers:  ");
        scanf("%f %f",&a,&b);
        if (b == 0.0)
            printf("The ratio is undefined¥n");
        else {
            ratio = a / b;
            printf("The ratio is %f ¥n",ratio);
        }
        printf("Press 'q' to quit, any other key to continue");
    } while (getch() != 'q');
}
```

このプログラムは比を計算したあと，どれかキーを押すようにうながします。そこで***q***を押すと，プログラムの最後の式が偽になってループが終了します。***q***以外のキーを押すと，この式は真になってループは繰り返されます。

do...whileループの一般的な形式は，次のようになります。

```
do  文  while (式);
```

while ループとの大きな違いは，do...while ループの中の文は少なくとも1回は必らず実行されるという点です。do...while ループは，違いが1つあることを除けば Pascal の **repeat... until** と同様な働きをします。違いというのは，**repeat** ループは条件が真になるまで(until)実行するのに対して，do...while は条件が真である間（while）実行するという点です。

関数

これまで，コードを条件によって実行したり，繰り返して実行したりする方法を見てきました。さてそれでは，異なるデータで，あるいはプログラムの異なる場所で，同じ一連の命令を実行したいときにはどうすればよいでしょうか?「そうした一連の文を**サブルーチン**にまとめ，必要に応じてこれを呼び出す」というのが解答です。

Cでは，すべてのサブルーチンは**関数**として存在します。原則としては，すべての関数はなんらかの値を返すことになっていますが，実際には返す値が無視される関数も数多くあります。また，Cの最近の定義では（ANSI の標準化案や Turbo C を含め），void という型で関数を宣言することによって，関数がなにも値を返さないことを指定できるようになっています。

関数は，**宣言**することと**定義**することができます。関数を宣言すると，プログラムの他の部分でもその関数を知ることができるので，他の関数が（**main** も含めて）その関数を呼び出すことが可能になります。関数を定義するときに，関数自身の実際のコード（その関数が何をするのか）を与えます。例題として，RATIO.C を書き直した次のプログラムを考えてみましょう。

```
#include <stdio.h>
#include <conio.h>

/*  関数の宣言  */
void  get_parms(float *p1, float *p2);
float get_ratio(float dividend, float divisor);
void  put_ratio(float quotient);

const float  INFINITY = 3.4E+38;

/*  main関数：プログラムの開始位置  */
main()
{
    float  a,b,ratio;

    do {
        get_parms(&a,&b);                             /*  引数を得る    */
        ratio = get_ratio(a,b);                       /*  比を計算する  */
        put_ratio(ratio);                             /*  答を表示する  */
        printf("Press 'q' to quit, any other key to continue ");
    } while (getch() != 'q');
}
/*  main終了  */

/*  関数の定義  */

void  get_parms(float *p1, float *p2)
{
    printf("Enter two numbers:  ");
    scanf("%f %f",p1,p2);
}

float get_ratio(float dividend, float divisor)
{
    if (divisor == 0.0)
        return (INFINITY);
    else
        return (dividend / divisor);
}

void  put_ratio(float ratio)
{
    if (ratio == INFINITY)
        printf("The ratio is undefined¥n");
    else
        printf("The ratio is %f ¥n",ratio);
}
```

プログラムの分析

プログラムの最初の3行では，関数の宣言を行なっています。エラーチェックのために，引数の個数と型，および関数の型を宣言するのがこの部分の目的です。

次の行では，INFINITY という浮動小数点の定数を定義しています(定数の名前を大文字にするのはCの慣習になっています)。この定数は非常に大きな正の値（float 型では最大値に近い値）を持っており，0による除算が行なわれたことを示すフラグ（標識）として用いられます。この定数は，ここ（すべての関数の外側）で定義されているので，すべての関数（**main** も含めて）の中から“見える”（参照できる）ようになっています。

次に来るのが **main** 関数で，これがプログラムの本体です。Cのプログラムには，すべて **main** という名前の関数があります。プログラムの実行が始まると，まず **main** 関数が呼び出され，そこからすべてのことが執り行われていくわけです。**main** 関数の実行が終わるとプログラムは終了し，Turbo C に戻ります（MS-DOS のプロンプトから起動した場合は，MS-DOS に戻ります)。

main 関数はプログラムの中のどこに置いてもかまいませんが，たいていはプロトタイプや他のグローバル宣言のすぐ後に，最初の関数として置かれます（プロトタイプ，グローバル宣言については，この後で説明します)。こうすると，**main** 関数を見つけやすくなり，プログラム全体の機能の文書化が楽になります。

main 関数の後ろには，プロトタイプで宣言された3つの関数（*get _parms*，*get _ratio*，*put _ratio*）の実際の定義が続きます。各関数の定義について見ていきましょう。

get _parms 関数

get _parms 関数は値を返さないので，void 型と定義しています。この関数の目的は，2つの値を読み込んで，どこかに格納しておくことです。どこへでしょうか。*get _parms* 関数には2つの引数を渡さなければなりません。この引数は値が格納される場所のアドレスです。したがって2つの引数は，float 型ではなく float 型を指すポインタになっています。言いかえると，これは float 型変数のアドレスということになります。

これが *get _parms* に渡すものです。**main** の中で *get _parms* を呼び出すときの引数は，*a* と *b* ではなく *&a* と *&b* です。また，*get _parms* の中の **scanf** の呼び出しでは，*p1* と *p2* の前にアドレス of 演算子 (&) はついていません。これは，*p1* と *p2* はすでにアドレス (*a* と *b* のアドレス）になっているからです。

get _ratio 関数

get _ratio 関数は，受け取った float 型の2つの値（*dividend*, *divisor*）から比を計算して，その結果の値（float 型）を返します。返す値は，*divisor*（除数）が0かどうかで異なります。*divisor* が0であれば *get _ratio* は *INFINITY* を返し，0でなければ実際の比を返します。return 文の形式に注目してください。

put _ratio 関数

put _ratio 関数は値を返さないので，void 型になっています。引数は1つ（*ratio*）だけで，画面に何を出力するかを決めるために使われます。*ratio* が *INFINITY* と等しければ未定義（undefined）とみなされ，そうでなければ *ratio* の値が出力されます。

グローバル宣言

あらゆる関数（**main** も含めて）の外側で宣言される定数，データ型，および変数は，宣言された時点から**グローバル**であるとみなされます。これは，宣言以降のプログラム全体にわたってあらゆる関数がこれを使用することができるという意味です。もし *INFINITY* の宣言をプログラムの最後に持っていくと，コンパイラエラーが2つ出るはずです。*get _ratio* 関数と *put _ratio* 関数の中で，Undefined symbol 'INFINITY' ...（宣言していない識別名を使用している）というエラーメッセージが出るでしょう。

関数の宣言

関数を宣言するには，2つのスタイルがあります。"古典"スタイルと"モダン"スタイルです。古典スタイルは，C 言語の文献やプログラムに多く見られるもので，次のような形式になります。

型　関数名();

これは，関数の名前と，関数が返す値のデータ型を指定しています。引数に関する情報はいっさいないのでエラーチェックは行なわれません。RATIO.C の関数宣言を古典スタイルで書きなおすと次のようになります。

```
void  get_parms();
float get_ratio();
void  put_ratio();
```

モダンスタイルは**関数プロトタイプ**として知られており，ANSI 標準化案の構文を用いたものです。モダンスタイルでは，引数に関する情報が付け加えられます。

```
型  関数名 (引数情報, 引数情報, etc.);
```

この中の**引数情報**は，以下の3つの書式のいずれかです。

```
型
型  引数名
...
```

つまり，各々の引数に対して，データ型だけを指定することもでき，データ型とともに名前を指定もしてもよいということです。関数が受け取る引数の個数が一定でない場合には，最後の引数としてピリオドを3つ並べた省略符号（...）を使うことができます。

推薦できるのはこのモダンスタイルの方です。このスタイルを使えば，コンパイルの際に，実際に関数を呼び出しているところで引数の個数と型がチェックできるからです。また可能な場合には，コンパイラが引数の型を適切な型に変換することもできます。RATIO.C の関数宣言が関数プロトタイプです。関数プロトタイプについては，第11章と第12章でくわしく説明しています。

関数の定義

関数の宣言と同様に，定義においても"古典"と"モダン"の2つのスタイルがあります。

古典スタイルの関数定義の書式は次のようになります。

```
型  関数名 (引数名)
引数の定義;
{
    ローカルな宣言;
    文;
}
```

モダンスタイルの書式では，引数の定義が，関数名の後ろのカッコの中に移動します。

```
型  関数名 (引数情報, 引数情報, etc.)
{  ...  }
```

引数情報には，その引数に関するすべての情報が含まれます。**型修飾子**と**識別名**です。これによって，関数定義の最初の行は，1つの例外を除けば，対応する関数のプロトタイプ

と同じになります。例外というのは，ここでは定義に続くセミコロン（;）がないという点です（関数のプロトタイプは必らずセミコロンで終わります）。たとえば，*get _parms* 関数を古典スタイルで書くと，次のようになります。

```
void  get_parms(pl, p2)
float *pl; float *p2;
{ ... }
```

また，モダンスタイルでは次のようになります。

```
void  get_parms(float *pl, float *p2)
{ ... }
```

関数（**main** も含めて）の中で宣言されたもの（定数，データ型，変数）は，その関数の中でのみ“見える”（つまり参照できる）という点を憶えておいてください。このような関数の中で行なわれる宣言をローカルな宣言と呼びます。なお，C では関数定義のネスト（入れ子）は許されていません。Pascal などとは違って，関数の中で別の関数を宣言することはできないわけです。

また関数は，プログラム内にどのような順序で置いてもよく，プログラム全体においてグローバルとみなされます。ただし，関数が定義または宣言される前にその関数を使用するときには注意が必要です。コンパイラは，それまでに見たことのない関数に出会うと，その関数は int 型の値を返すものとみなします。後でその関数が，int とは違う値（char * など）を返すことが定義されていると，エラーが出ることになります。

コメント

プログラムの中に注釈を入れて，変数や関数あるいは文などが何を意味しているのかが思い出せるように（また他の人が見てわかるように）しておきたいことがあります。こうした注釈を**コメント**と呼びます。他のほとんどのプログラミング言語と同様に，C でもプログラムにコメントを入れることができます。

コメントは，スラッシュ-アスタリスク（/*）で始まります。コンパイラは，そこから次に来るアスタリスク-スラッシュ（*/）までのすべてのテキストを無視します。

コメントは，次のように複数行にまたがってもかまいません。

```
/* This is a long
   comment, extending
   over several lines. */
```

RATIO.C の拡張版は，コメントの例にもなっています。

まとめ

この章では，プログラミングの7つの基本要素について紹介し，これらを Turbo C ではどうやって使うのかというところまでお話してきました。

次の第7章では，これらについてさらに詳しく述べ，また Turbo C の他の機能についても触れています。

第 7 章

続：Turbo C でのプログラミング

前の章では，Turbo C を動かす味わいを賞味していただきました。食欲もさらにそそられていることでしょう。もうすでに，C プログラミングの深遠な奥義を掘りさげていく準備はととのっています。さあ，進んでいきましょう。

この章では……

この章では，C 言語の以下のような範囲をカバーしています。

■データ構造（ポインタ，配列，構造体）

■ switch 文

■ return, break, continue, goto, 条件演算子(? :)などの実行の流れを制御する文

■ C のプログラムスタイル，特に新たに拡張された点について

■ C プログラマが陥りやすい落し穴

データ構造の概観

第6章では，基本的なデータ型（整数型，浮動小数点型，文字型など）について説明しました。この節では，こうした基本的な要素を組み合わせて**データ構造**（データ要素の集合）を作り出す方法について説明しますが，その前に，Cの重要な概念である**ポインタ**について見ておきましょう。

ポインタ

今まで取り上げてきたほとんどの変数はデータを保持しており，これはプログラムが操作する実際の情報でした。しかし，データの値そのものではなく，データが置かれている場所をたどっていきたい場合があります。これを行なうために，ポインタが必要になるのです。

アドレスやメモリという概念にあまりなじみのない方のために，ちょっと説明をしておきましょう。コンピュータは，プログラムとそれに関係するデータをメモリ（たいていはRAMと呼ばれ，Random Access Memoryの略です）の上に保持するようになっています。最も低いレベルで見ると，メモリは**ビット**の集合から成りたっています。ビットは極小の電子回路で，コンピュータの電源が入っているあいだは，2つの値のどちらかを“記憶”することができます。この2つの値は，通常0あるいは1と解釈されます。

ビットが8個集まったものが**バイト**です。バイトの集合にも，こうした名前のついているものがいくつかあります。一般に，バイトが2個で**ワード**，バイトが4個で**ロングワード**と呼ばれています。また，パーソナルコンピュータでは16バイトを**パラグラフ**と呼んでいます。

コンピュータのメモリ内の各バイトには，固有の番地がつけられています。町内の家々にそれぞれ番地がついているようなものです。実際の町では，隣りあう家の番地が順番になっていないこともありますが，バイトの番地（普通**アドレス**と呼びます）は必らず順番に並んでいます。*N* 番地というアドレスのバイトがあれば，そのすぐ前は *N*-1番地であり，すぐ後ろは *N*+1番地というようになっているわけです。

ポインタというのは，なんらかのデータが格納されている場所，つまりアドレスを持っている変数のことで，データそのものではありません。ポインタはどんなことに役立つのでしょうか。まず第一に，1つのポインタで，異なるデータや異なるデータ構造を指し示す（pointする）ことができます。ポインタが持っているアドレスを変えてやることによっ

て，いろいろな場所にある情報を操作すること（代入，参照，変更）ができるのです。たとえば，たった1つのポインタで，構造体という異なる型のデータが並んだリストをたどっていくことができます。

第2には，ポインタを使えば，プログラムの実行中に新しい変数を作ることも可能になります。C では，プログラムが必要とするメモリをバイト単位で確保することができ，確保されたメモリのアドレスはポインタにおさめることができます。これは，動的割り当て（dynamic allocation）として知られる機能で，これを使うと，使用するコンピュータのメモリの大きさにプログラムを適合させることができます。

第3に，1つのポインタを使って，データ構造（配列，文字列，構造体など）の中の異なる場所にアクセスすることができます。1つのポインタが実際に指し示すことができるのは，メモリ上のただ1ヶ所（セグメントとオフセットによって定まる）だけですが，ポインタが持っているアドレスをずらしていけば，連続したバイトの並びをアクセスすることができるわけです。

ポインタが便利なものだということは，もう納得していただけたでしょう。では，どのように使うのでしょうか? まず，ポインタの宣言をしなければなりません。次のようなプログラムを考えてみてください。

```
#include <stdio.h>

main()
{
     int ivar, *iptr;

     iptr = &ivar;
     ivar = 421;
     printf("location of ivar: %p¥n",&ivar);
     printf("contents of ivar: %d¥n", ivar);
     printf("contents of iptr: %p¥n", iptr);
     printf("value pointed to: %d¥n",*iptr);
}
```

このプログラムの **main** では，*ivar* と *iptr* という2つの変数を宣言しています。最初の *ivar* は整数の変数，つまり int 型の値を持っています。もう1つの *iptr* は整数の変数へのポインタ，つまり int 型の値が置かれているアドレスを持っています。宣言の中で，*iptr* の前にアスタリスク（*）がついていますが，このアスタリスクによって *iptr* がポインタであることを指定しています。C では，このアスタリスクは**間接演算子**と呼ばれています。

main では，以下のように代入が行なわれます。

■ *ivar* のアドレスが *iptr* に代入される
■整数値421が *ivar* に代入される

ivar のアドレスを得るために，前の章で出てきたアドレス of 演算子（&）を使っています。

上のプログラムを入力して実行してみてください。次のような出力が得られるでしょう（表示されるアドレスはこれとは異なっていることもあります）。

```
location of ivar: 166E
contents of ivar: 421
contents of iptr: 166E
value pointed to: 421
```

最初の2行は，*ivar* のアドレスとその内容を示しています。また3行目は，*iptr* が持っているアドレスを示しています。おわかりのように，これは *ivar* のアドレスであり，プログラムが *ivar* を作成したメモリ上の場所です。最後の行の値は，その場所におさめられているデータで，すでに *ivar* に代入されているデータとまったく同じものです。

printf の3番目の呼び出しでは，*iptr* という式が，その内容である *ivar* のアドレスを得るために使われていることに注意してください。そして，最後の **printf** では，そのアドレスにおさめられているデータを取り出すために，＊ *iptr* という式が使われています。

先程のプログラムをすこし書きなおしたのが，下のプログラムです。

```
#include <stdio.h>

main()
{
    int ivar, *iptr;

    iptr = &ivar;
    *iptr = 421;
    printf("location of ivar: %p¥n",&ivar);
    printf("contents of ivar: %d¥n", ivar);
    printf("contents of iptr: %p¥n", iptr);
    printf("value pointed to: %d¥n",*iptr);
}
```

ここでも，*ivar*のアドレスを*iptr*に代入していますが，421という値を，*ivar*ではなく＊ *iptr* に代入しています。結果はどうなるでしょうか?じつは前のプログラムとまったく同じになります。なぜなら，＊ iptr ＝ 421と ivar ＝ 421という2つの文は，同じことを意味しているからです。つまり，*ivar* と＊ *iptr* はメモリ上の同じ場所を参照しているので，どちらの式も421という値を同じ場所に代入しているわけです。

メモリの動的割り当て

先程のプログラムを，さらにもうすこし書きなおしたのが下のプログラムです。

```
#include <stdio.h>
#include <alloc.h>

main()
{
    int *iptr;

    iptr = (int *) malloc(sizeof(int));
    *iptr = 421;
    printf("contents of iptr: %p¥n", iptr);
    printf("value pointed to: %d¥n",*iptr);
}
```

ここでは，変数 *ivar* は宣言されていません。そのかわり，**malloc** という関数が返す値を *iptr* に代入しています。関数 **malloc** はヘッダファイル alloc.h の中で定義されています(このため，最初に# include 指令で alloc.h を取り込んでいます)。そして数値421は，＊ *iptr*，すなわち *iptr* が指し示すアドレスに代入されています。このプログラムを実行すると，おそらく前のプログラムとは異なる *iptr* の値が得られるでしょうが，＊ *iptr* の値は421のままです。

では，この文が何をしているのか分解して考えてみましょう。

```
iptr = (int *) malloc(sizeof(int));
```

■ sizeof(int) という式は，int 型の変数が必要とするバイト数を返します。

■関数 **malloc**(*num*)は，利用可能な(未使用)のメモリから，連続する *num* バイト分の領域を確保し，その領域の先頭のアドレスを返します。

■(int ＊)という式は，領域の先頭のアドレスを int 型へのポインタとして扱うことを意味します。これは**型キャスト**と呼ばれるものです。Turbo C の場合は，型キャストはなくてもかまいません。しかし，多くのCコンパイラでは型キャストが必要であり，これがないと“このポインタの代入は可搬性が低い”というような警告メッセージが出力されます。

■最後に，このアドレスが *iptr* に格納されます。これは，整数型変数を動的に作り出したことを意味しており，この整数型変数は＊ *iptr* として参照することができます。

以上の説明をまとめると，「int 型の変数1個が必要とするサイズのメモリ領域を，コンピュータのメモリ上に割り当て，その領域の最初のアドレスを，int 型へのポインタである

iptr に代入する」ということになります。

以上の操作がすべて必要だったのでしょうか?もちろん必要です。なぜか?このような操作を行なっていなければ，*iptr* がメモリ上の未使用の領域を指しているという保証は何もありません。*iptr* はなんらかのアドレス値を持つでしょうが，そのアドレスは別の目的に使用されているかもしれません。ポインタを使うときに守らなければならないルールはたいへん簡単です。それは，「ポインタを使うときには必らず，前もってそのポインタにアドレスを代入しておきなさい」ということです。上の例でいえば，*iptr* にアドレスを代入していないのに，＊ *iptr* に整数値を代入してはいけないということです。

ポインタと関数

第6章では，関数の引数の宣言方法について説明しました。関数の中で値を変更したい仮引数になぜポインタを使うのか，もうすでにおわかりの方もいるかもしれません（関数の定義をする際に，その関数が受け取る引数として定義するものを仮引数と呼びます。また関数が呼び出される際に，その関数に渡される実際の引数を実引数と呼びます)。

この点について，次のような関数を考えてみましょう。

```
void swap(int *a, int *b)
{
    int temp;
    temp = *a; *a = *b; *b = temp;
}
```

この関数 *swap* では，int 型のデータを指すポインタ *a* と *b* が仮引数として宣言されています。これは，整数型変数の（値そのものでなく）アドレスが関数に渡されることを意味しています。変更は，渡されたアドレス位置に存在するデータに対して加えられます。

次に，*swap* を呼び出す **main** 関数を見てみましょう。

```
main()
{
    int i,j;

    i = 421;
    j = 53;
    printf("before: i = %4d  j = %4d¥n",i,j);
    swap(&i,&j);
    printf("after: i = %4d  j = %4d¥n",i,j);
}
```

このプログラムで，確かに *swap* が *i* と *j* の値を入れかえていることがわかります。このプログラムは，以下のように書きかえることもできると考えられるでしょう。

```
main()
{
    int i,j;
    int *a,*b,temp;

    i = 421;
    j = 53;
    printf("before: i = %4d  j = %4d¥n",i,j);
    a = &i;
    b = &j;
    temp = *a; *a = *b; *b = temp;
    printf("after: i = %4d  j = %4d¥n",i,j);
}
```

このプログラムも，もちろん前と同じ結果を出力します。swap(&i,&j)という文は，*swap*関数を呼び出し，2つの実引数（*&i*，*&j*）の値を，2つの仮引数（*a*，*b*）に代入し，*swap*関数内の文を実行するわけです。

ポインタの演算

*iptr*が，（1つだけではなく）3つの整数を指し示せるようにしたいとしたら，どうすればよいのでしょうか。

次のプログラムは，その解答の一つです。

```
#include <stdio.h>
#include <alloc.h>

main()
{
    #define NUMINTS 3
    int  *list,i;

    list = (int *) calloc(NUMINTS,sizeof(int));
    *list = 421;
    *(list+1) = 53;
    *(list+2) = 1806;
    printf("list of addresses: ");
    for (i = 0; i < NUMINTS; i++)
        printf("%4p ", (list+i));
    printf("¥nlist of values    : ");
    for (i = 0; i < NUMINTS; i++)
        printf("%4d ", *(list+i));
}
```

こんどは，**malloc**のかわりに**calloc**関数を使用しています。**calloc**は2つの引数をとります。最初の引数はメモリ上に確保する要素の個数，2番目は要素1つ当たりに必要なバイト数です。したがって，変数*list*は，int型の3つの変数を記憶するための6バイト（=3×2）の領域を指すことになります。

その後の3行の文をよく注意して見てください。最初の文はもうおなじみの＊ list ＝ 421です。これは，「*list* が示しているアドレスにある int 型変数に421を代入せよ」ということです。

その次の文＊(list＋1)＝ 53，これが重要ですのでよく理解してください。ちょっと見ると，「*list* が示すアドレスの1バイト先にある int 型変数に53を代入せよ」というように読み取れます。しかし，もしそうだとすると，int型の変数は 2 バイトの大きさがあるので，＊ list ＝ 421で代入したメモリの2バイト目に，今度代入する値の1バイト目が重なってしまい，前に代入した値を壊してしまうことになります。

心配はいりません。C コンパイラはもう少しインテリジェントです。コンパイラには，*list* は int 型変数のポインタであるとわかっているので，list＋1は list＋(1＊ sizeof(int))のことであると判断して，53という値は421とは重ならずにメモリに格納されます。

同様に，list＋2は list＋(2＊ sizeof(int))というアドレスを参照しているので，1806も前の2つの値に影響を与えることなく無事に代入されます。

一般に，「***ptr*＋*i* は，*ptr*＋(*i* ＊ *sizeof*(*int*))というアドレスを意味する**」ということになります。

このプログラムを入力して実行してみてください。次のような出力が得られるでしょう。

```
list of addreses:  06AA 06AC 06AE
list of values :    421   53 1806
```

2バイトおきのアドレスに，3つの値が重複することなく格納されています。

ここで説明したことをまとめてみましょう。データ型 *type* へのポインタ *ptr* を使用するとき，

- sizeof(type)は type 型変数が必要とするバイト数を返し，
- ptr＋i は，ptr＋(i ＊ sizeof(type))というアドレスを意味する，

ということになります。

配列

Cも含めた多くの高水準言語では，**配列**を使用することができます。配列は，与えられた型のデータの索引つきのリストと考えることができます（この索引は，通常添字と呼ばれます)。たとえば，前のプログラムを配列を使って書きかえると次のようになります。

```
#include <stdio.h>

main()
{
    #define NUMINTS 3
    int  list[NUMINTS],i;

    list[0] = 421;
    list[1] = 53;
    list[2] = 1806;
    printf("list of addresses: ");
    for (i = 0; i < NUMINTS; i++)
        printf("%p ", &list[i]);
    printf("¥nlist of values   : ");
    for (i = 0; i < NUMINTS; i++)
        printf("%4d ", list[i]);
    printf("¥n");
}
```

int list[NUMINTS]という式は，*list* を int 型の配列として宣言し，int 型の変数3個分のデータ領域を確保しています。最初の変数は *list*[0]で参照でき，2番目と3番目の変数はそれぞれ *list*[1]，*list*[2]で参照できます。

配列の宣言は，一般に次のような形で行ないます。

```
type name[size];
```

type はなんらかのデータ型，*name* は配列につける名前，*size* は配列 *name* に含まれる *type* 型の要素の個数です。最初の要素は *name*[0]となり，最後の要素は *name*[*size*-1]となります。配列全体としてはメモリ上に，*size* ＊(sizeof(*type*))バイトの領域を占めます。

配列とポインタ

ここまでの説明で，配列とポインタの間には密接な関係がある，ということに気づいた方もいることと思います。前の配列を使ったプログラムを実行すると，次のような，ポインタを使った場合とよく似た結果が出力されます。

```
list of addresses: 163A 163C 163E
list of values    :  421   53 1806
```

最初の要素のアドレスが異なっていますが，違いはそこだけです。実際にも，配列名はポインタとして扱うことができ，ポインタにも配列のような添字をつけることができます。

次に示す関係はたいへん重要なので，意味をよく考えてみてください。

```
(list + i) == &(list[i])
*(list + i) == list[i]
```

この2つの式は，左辺と右辺が同じものであることを意味しています。*list* をポインタと宣言しているか，配列と宣言しているかにかかわらず，どちらの式も使用できます。

list をポインタとして宣言した場合と，配列として宣言した場合の違いは，メモリの割り当てに関する点だけです。*list* を配列として宣言すると，必要なメモリ領域はプログラムが自動的に確保します。ポインタとして宣言した場合には，**calloc** あるいは同種の関数を使って明示的に必要な領域を作り出すか，すでに確保されている領域のアドレスをポインタに代入しなければなりません。

配列と文字列

前の章で，少しばかり異なる2つの方法による文字列の宣言(文字へのポインタと文字配列）についてお話ししました。いまは，その違いについてより深く理解することができるでしょう。

文字列を char 型の配列として宣言すると，必要な大きさの領域がメモリ上に割り当てられます。char 型へのポインタとして宣言すると，領域は割り当てられないので，(**malloc** あるいは同じような動作をする関数を使って）ユーザ自身の手で割り当てるか，すでに存在している文字列のアドレスをそのポインタに代入しなければなりません。これについての例は，この章の「**Cプログラミングの落し穴**」の節に出てきます。

多次元の配列

Cでは，多次元配列も使うことができます。その宣言は次のようにして行ないます。

```
type  name[size1] [size2] ... [sizeN];
```

次のプログラムを見てみましょう。これは，2つの配列を初期化し，その2つの配列の積を計算しています。

```
#include <stdio.h>

main()
{
    int a[3][4] = { {  5,  3,-21, 42},
                    { 44, 15,  0,  6},
                    { 97,  6, 81,  2}  };
    int b[4][2] = { { 22,  7},
                    { 97,-53},
                    { 45,  0},
                    { 72,  1} };

    int c[3][2],i,j,k;

    for (i = 0; i < 3; i++) {
        for (j = 0; j < 2; j++) {
            c[i][j] = 0;
            for (k = 0; k < 4; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
    }
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 2; j++)
            printf("c[%d][%d] = %d ", i, j, c[i][j]);
        printf("¥n");
    }
}
```

このプログラムでは2つの点に注目してください。1つは配列の初期化の方法で，{...}をカンマで区切り，それをネストして2次元の配列を初期化しています。もう1つは，配列の添字をそれぞれ大カッコ[ ]で囲んでいる点です。

言語によっては，配列の添字を[*i*,*j*]と書くものもあります。Cでもこの書き方は文法的には間違いではありませんが，意味としては[*j*]と判断されてしまいます。なぜなら，Cではカンマは逐次演算子として扱われるので，[*i*,*j*]という式では，まず*i*を評価し，次に*j*を評価して，全体の値は最後に評価された*j*と等しくなってしまうのです。ですから，添字ごとに大カッコで囲むことを忘れないでください。

多次元配列では，データは**行-列の順**に格納されます。つまり，格納された配列の要素を先頭から順にたどっていくと，後ろに書かれている添字から先に数字が増えていきます。たとえば，*arr*[3][2]と宣言すると，*arr*の各要素は次のような順番で格納されます。

```
arr[0][0]
arr[0][1]
arr[1][0]
arr[1][1]
arr[2][0]
arr[2][1]
```

3次元以上の配列の場合も同様です。

配列と関数

配列を，引数として関数に渡すと何が起こるでしょうか?

次の関数を見てください。これは，int 型の配列の中から最小の要素を見つけだし，その**添字**を返す関数です。

```
int  lmin(int list[], int size)
{
    int  i, minindx, min;

    minindx = 0;
    min = list[minindx];
    for (i = 1; i < size; i++)
        if (list[i] < min) {
            min = list[i];
            minindx = i;
        }
    return(minindx);
}
```

この例は，C という言語の偉大な力の1つを物語っています。コンパイル時には，*list*[]のサイズはわかっていなくてもよいのです。コンパイラは，*list*[]が配列の先頭のアドレスであると認識しますが，その配列がどこで終わるのかは気にかけません。*lmin* を呼び出すプログラムは，次のように書くことができます。

```
#include <stdio.h>

main()
{
    #define VSIZE 22
    int  i, vector[VSIZE];

    for (i = 0; i < VSIZE; i++) {
        vector[i] = rand();
        printf("vector[%2d] = %6d\n", i, vector[i]);
    }
    i = lmin(vector,VSIZE);
    printf("minimum: vector[%2d] = %6d\n", i, vector[i]);
}
```

ここで問題です。*lmin* に引数として渡されるものの実体はなんでしょうか?　解答:配列 *vector* の最初の要素のアドレスです。これは，*lmin* の中で配列 *list* の要素の値を変更すると，配列 *vector* の要素も同時に変更されるということを意味しています。たとえば，次のような関数を書くことができます。

```
void setrand(int list[], int size)
{
    int i;
    for (i = 0; i < size; i++) list[i] = rand();
}
```

この関数を，**main** の中で setrand(vector,VSIZE) として呼び出せば，*vector* を初期化することができます。

それでは，多次元配列を関数に渡すときはどうするのでしょうか。1次元配列と同じようなフレキシビリティが得られるでしょうか。*setrand* が2次元配列に対して機能するように変更したいとしましょう。

次のように書きかえることになるでしょう。

```
void setrand(int matrix[][CSIZE],int rsize)
{
    int i,j;
    for (i = 0; i < CSIZE; j++)
        matrix[i][j] = rand();
    }
}
```

CSIZE はグローバル定数で，2次元配列の2次元目の大きさを定めています。つまり，*setrand* に渡す2次元配列の2次元目の大きさは，*CSIZE* でなければならないのです。

しかしながら，解決策はもう1つあります。*setrand* に *matrix*[15][7]を渡す場合を考えてみてください。*setrand* が元の定義のままでも (setrand(int list[], int size))，次のようにして呼び出すことができます。

```
setrand(matrix, 15*7);
```

配列 *matrix* は，*setrand* の中では105 (＝15×7) の1次元配列として扱われますが，すべて完全に動作します。

構造体

配列とポインタによって，同一のデータ型の項目からなるリストを作ることができました。では，異なるデータ型の項目からなるデータ構造を作るにはどうすればよいのでしょうか。こういう場合に，**構造体**を宣言します。

構造体は，異なる型のデータが集まってできたデータ構造です。たとえば，“星”に関する情報として，‘名前’，‘スペクトルクラス’，‘位置の座標’などの項目を取り扱いたいとする

と，次のような構造体を宣言することができるでしょう。

```
typedef struct {
    char   name[25];
    char   class;
    short  subclass;
    float  decl,RA,dist;
} star ;
```

これは，*star* という型名の構造体を定義しています。構造体の中で宣言された各変数（*name*，*class*，・・・）を，その構造体の**メンバ**と呼びます。

これが宣言されていれば，つまりプログラムの最初の方にこの宣言を書いておけば,，それを使って star 型の構造体変数を宣言することができます。

```
main()
{
    star mystar;     /*  star型の構造体変数 mystar の宣言  */

    strcpy(mystar.name,"Epsilon Eridani");
    mystar.class     = 'K';
    mystar.subclass =  2;
    mystar.decl      =  3.5167;
    mystar.RA        = -9.633;
    mystar.dist      =  0.303;

    /*  main関数のその他の部分  */
}
```

構造体変数の個々のメンバを参照するには，構造体変数名の後にピリオドとメンバ名を続けて書きます。たとえば，構造体 *varname* のメンバ *memname* は *varname.memname* と記述します。*varname.memname* は *memname* と同じ型の変数名として扱うことができ，同じ型の変数に対して可能なすべての操作を行なうことができます。

構造体とポインタ

他のデータ型と同様に，構造体を指すポインタを宣言することができます。この機能は，線形リストなどの動的なデータ構造を組み立てる際に不可欠なものです。C では構造体を指すポインタはしばしば使用されるので，ポインタが指す構造体のメンバを参照するための特別な記号が用意されています。

先程のプログラムを，次のように書きかえてみましょう。

```
#include <alloc.h>

main()
{
    star *mystar;

    mystar = (star*) malloc(sizeof(star));
    strcpy(mystar -> name,"Epsilon Eridani");
    mystar -> class = 'K';
    mystar -> subclass = 2;
    mystar -> decl = 3.5167;
    mystar -> RA = -9.633;
    mystar -> dist = 0.303;

    /*  main関数のその他の部分  */
}
```

こんどは，*mystar* は *star* 型の変数ではなく，*star* 型への**ポインタ**として宣言しています。そして，*mystar* のための記憶領域を，**malloc** の呼び出しを通じてメモリ上に割り当てています。これで，*ptr->memname* という式で *mystar* が指す構造体のメンバを参照することができます。->という記号は，「～によって指し示される構造体のメンバ」という意味で，（* *ptrname*）*.memname* という記法の省略形といえます。

switch 文

if..else if..else if..という長い構文を書いてしまうことがあるかもしれません。次の関数を見てください。

```
#include <ctype.h>

do_main_menu(short *done)
{
    char  cmd;

    *done = 0;
    do {
        cmd = toupper(getch());
        if (cmd == 'F') do_while_menu(done);
        else if (cmd == 'R') run_program();
        else if (cmd == 'C') do_compile();
        else if (cmd == 'M') do_make();
        else if (cmd == 'P') do_project_menu();
        else if (cmd == 'O') do_option_menu();
        else if (cmd == 'E') do_error_menu();
        else handle_others(cmd,done);
    } while (!*done);
}
```

こうした構文はしばしば現われるものですが，Cにはこのような場合のための特別な制御構造があります。switch 文がそれです。上の関数を，switch 文を使って書きかえると次のようになります。

```
#include <ctype.h>

do_main_menu(short *done)
{
    char  cmd;

    *done = 0;
    do {
        cmd = toupper(getch());
        switch (cmd) {
            case 'F': do_file_menu(done); break;
            case 'R': run_program(); break;
            case 'C': do_compile(); break;
            case 'M': do_make(); break;
            case 'P': do_project_menu(); break;
            case 'O': do_option_menu(); break;
            case 'E': do_error_menu(); break;
            default : handle_others(cmd,done);
        }
    } while (!*done);
}
```

この関数はまずdo...whileループに入ります。このループは，1文字を読み込み，読み込んだ文字を大文字に変換し，変換した文字を *cmd* に代入し，*cmd* の値によってswitch文を実行する，という動作を繰り返します。ループは，*done* の値が（おそらくは *do _file _menu* 関数か *handle _others* 関数で）1になるまで繰り返されます。

switch文は *cmd* の値を取り出して，caseラベル（'F','R','C',...'E'）と順に比較していきます。一致するものがあれば，そのラベルのところから実行を始め，break文に出会うか，switch文の終わりに達するまで実行を続けます。一致するものがない場合は，switch文の中にdefaultというラベルがあればそこから実行し，なければswitch文全体を飛び越します。

switch文において，*cmd* は整数コンパチブル，言いかえれば容易に整数に変換できる値でなければなりません。つまり *cmd* に使用できるのはできるのは，char型，enum型，およびすべてのint型ということになります。float型やdouble型などの実数，ポインタ，文字列，データ構造（配列，構造体）は使えません。ただし，データ構造の要素で整数コンパチブルなものは使用できます。

cmd には，整数コンパチブルであればどんな式でも（定数，変数，関数呼び出し，およびこれらの組み合わせ）使用することができますが，caseラベルは定数でなければなりません。また，caseキーワード1つについては，値も1つしか指定できません。

do _main _menu 関数が**toupper**関数を使用しない，つまり *cmd* を大文字に変換しないとすると，switch文は次のように書かなければなりません。

```
switch (cmd) {
    case 'f' :
    case 'F' : do_file_menu(done);
               break;
    case 'r' :
    case 'R' : run_program();
               break;
    ...
```

このswitch文では，*cmd* の値が小文字fか大文字Fであれば，*do _file _menu* が実行されます。他のラベルについても同様です。

各ケースの処理の終わりにbreak文を置くことを忘れてはいけません。breakを書き忘れると，（次のbreak文に出会うまで）それ以降の文が実行されてしまいます。たとえば上のswitch文で，*do _file _menu* の後のbreakを取り除くと，**F**が入力されると *do _file _menu* が呼び出され，その後 *run _program* も呼び出されてしまいます。

ただし，次のプログラムのように，意図的にこのような流れを利用する場合もあります。

```
typedef enum { sun, mon, tues, wed, thur, fri, sat, } days;

main()
{
    days today;

    ...
    switch (today) {
        case mon:
        case tues:
        case wed:
        case thur:
        case fri: puts("go work!"); break;
        case sat: printf("clean the yard and ");
        case sun: puts("relax!");
    }
    ...
}
```

この switch 文では，*mon* から *fri* までの値に対しては，すべて同じ **puts** 文が実行され，その後の break 文によって switch 文が終了します。*sat* の場合は，**printf** 文が実行された後，puts(”relax!”)が実行されます。*sun* の場合は，最後の **puts** 文だけが実行されます。

実行の流れを制御する構文

制御構造内で使用したり，他の制御構造をシミュレートするための付加的な文がいくつかあります。return 文は，関数の途中でその関数を終了させます。break 文と continue 文はループの中で使用され，それ以降の文をスキップします。goto 文は，コード内でのジャンプを可能にします。また，条件演算子（?:）は，ある種の if...else 文をたった1行に圧縮してしまうことができます。

アドバイス:これらの制御構文は，使う前にもう一度よく考えてみましょう（もちろん，return を除いてですが）。使用せざるをえない場合もありますが，制御構文の力を借りなくても，問題をうまく解決できる方法が見つかる場合の方が多いものです。特に，goto はできるだけ避けるべきです。return，break，continue があれば，goto を使う必要はまずありません。

return 文

return 文には2つの大切な機能があります。まず第1に，関数が値を返すときには，return を使ってその値を呼び出しルーチンに渡してやらなければなりません。

```
int  imax(int a, int b)
{
    if (a > b)
        return(a);
    else
        return(b);
}
```

この関数では，受け取った2つの値の大きい方を返すために return 文を使用しています。

return 文の第2の用途は，関数が最後まで実行される前に関数を終了させる場合です。たとえば，関数を実行していく途中で関数の終了条件が成立してしまうような場合です。関数の残りの部分を if 文にまとめてしまうという方法もありますが，return 文を書くだけでも関数を終了させることができます。void 型の関数であれば，return だけを（(*value*)を書かずに）使って値を返さないことを明示することができます。

前に取り上げた *lmin* 関数を，return 文を使って書きかえてみましょう。

```
int lmin(int list[], int size)
{
    int i, minindx, min;

    if (size <= 0)
        return(-1);
        ...
}
```

引数 *size* がゼロ以下の場合は，配列 *list* の中には何もないことになるので，ただちに return を呼んで関数を終了させています。このとき，エラーを示す値として-1を返している点に注意してください。配列の添字が-1になることはありえないので，関数を呼び出した側ではこの値（-1）から，なんらかのエラーが起きたことがわかるわけです。

break 文

ループの終わりに達する前に，すぐにまた簡単にループの外に出たい場合があります。次のプログラムを考えてみましょう。

```
#define LIMIT 100
#define MAX   10

main()
{
    int  i,j,k,score;
    int  scores[LIMIT][MAX];

    for (i = 0; i < LIMIT; i++) {
        j = 0;
        while (j < MAX-1) {
            printf("please enter score #%d: ",j);
            scanf("%d",&score);
            if (score < 0)
                break;
            scores[i][++j] = score;
        }
        scores[i][0] = j;
    }
}
```

if (score < 0) break;という文に注目してください。これは，ユーザが *score* に対して負の値を入力したら while ループは終了する，ということです。変数 *j* は，配列 *scores* の添字として使用されているほか，配列 *scores* の各行にあるデータ数のカウンタとしても使われています。この値は配列 *scores* の各行の最初の要素に格納されます。

前の節の switch 文における break の使い方を思い出されたかもしれません。switch の

場合には，break は switch 文から出るために使われていました。ここでは，ループの外に出てプログラムの実行を続けるために使われています。break 文は，switch 文の他に3種類のループ（for，while，do...while）の中で用いることができます。しかし，if...else 文の中や，関数本体の中では使用できません。

continue 文

ループの外側には出ずに，ループ内の残りの処理を飛び越して，ループの先頭から実行を再開したいという場合もあります。こうした状況では continue 文を使用します。

次のプログラムを見てください。

```
#define LIMIT 100
#define MAX   10

main()
{
    int  i,j,k,score;
    int  scores[LIMIT][MAX];

    for (i = 0; i < LIMIT; i++) {
        j = 0;
        while (j < MAX-1) {
            printf("please enter score #%d: ",j);
            scanf("%d",&score);
            if (score < 0)
                continue;
            scores[i][++j] = score;
        }
        scores[i][0] = j;
    }
}
```

これは，前のプログラムの break を continue に置きかえただけのものです。continue が実行されると，ループ内のそれ以降の文を飛び越して，while ループの先頭の式が再び評価されます。その結果，プログラムは break を使った場合とは異なる動きを示します。*score* に-1を入力すると，エラーが起きたものとみなされて，再び while ループの先頭に戻ります。変数 *j* はインクリメントされないので，前回と同じメッセージがもう一度表示されます。

goto 文

Cにも goto 文があります。書式は簡単で，goto label と書くだけです。*label* は特定の文に結びつけられた識別名です。goto 文の最も賢い使い方は，前の3つの文（return, break, continue)のどれかにその場をゆずることです。本当に goto 文でなければならないのかどうかよく考えるようにしてください。

条件演算子（? :)

ある条件に基づいて，2つの式（および，その結果として得られる値）のうちのどちらかを選択したいとします。

すでに見たように，普通これは次のような if...else 文で行ないます。

```
int  imin(int a, int b)
{
    if (a < b)
        return(a);
    else
        return(b);
}
```

if...else 文は，ある特殊な演算子にとってかわられるほど頻繁に用いられます。その演算子の書式は次のようになります。

```
式1 ? 式2 : 式3
```

これは，「もし**式1**が真であれば，**式2**を評価してその値を式全体の値とし，**式1**が偽なら**式3**を評価してその値を式全体の値とせよ」というように解釈されます。この構文を使うと，*imin* 関数を次のように書きかえることができます。

```
int  imin(int a, int b)
{
    return((a < b) ? a : b);
}
```

さらに，*imin* 関数はインラインマクロとして次のように書き直すことができます。

```
#define imin(a,b) ((a < b) ? a : b)
```

このようにすると，コンパイラはプログラム中に imin(e1,e2) という式を見つけるたび

に，((e1<e2) ? e1 : e2)で置き換えてコンパイルしていきます。これは，実際によく使用される一般的な手法です。インラインマクロにしておけば，*imin* 関数の引数 *a* と *b* のデータ型は int に固定されず，関係演算子<が使用できるものであればどんな型でも受け入れることができるのです。

ストリームとストリームI/O

ストリームとは?

ストリームは，Turbo C でデータの読み書きを行なう手法として最も移植性の高いものです。ストリームは，ファイルやハードウェアのデバイスといった低レベル機能を気にせずに，柔軟で効率的な入出力ができるよう設計されています。

ストリームとは，FILE 型のオブジェクト（stdio.h で定義されています）へのポインタを介して扱うファイルや物理デバイス（たとえばプリンタやモニタ）のことをいいます。FILE 型オブジェクトには，ストリームの現在位置，関連するバッファへのポインタ，エラーやファイルエンドインジケータなどのさまざまな情報が含まれています。

ユーザプログラムが，FILE 型のオブジェクトそのものを生成したりコピーしたりすることはありません。プログラムでは，**fopen** などの関数が返す FILE 型へのポインタを使用します。FILE 型ポインタと，MS-DOS のファイルハンドルを混同しないよう充分注意してください(ファイルハンドルは，MS-DOS の低レベル入出力や，UNIX と互換性のある入出力に使用します)。

ストリームの入出力を行なうには，まずストリームをオープンしなければなりません。ストリームをオープンすると，そのストリームは指定された MS-DOS ファイルあるいはデバイスに接続されます。ストリームをオープンするルーチンには，**fopen**，**fdopen**，**freopen** があります。ストリームをオープンする際には，そのストリームに対して読み込みを行なうのか，書き出しを行なうのか，あるいはその両方なのかを指定します。またそのストリームでは，データを**テキスト**として扱うか，**バイナリ**として扱うかについても指定します。テキストかバイナリかの指定が問題となるのは，C のストリーム入出力と DOS のテキストファイルとの間に一部互換性のない部分があるためです。

テキストストリームとバイナリストリーム

テキストストリームは，たとえば Turbo C のエディタで作成したような普通の DOS のテキストファイルに対して使用します。C のストリーム入出力では，テキストファイルの行の区切りを改行文字 (ASCII コード0x0A) 1文字とみなします。ところが MS-DOS のテキ

ストファイルは，行の区切りは復帰（0x0D）と改行（0x0A）の2文字でディスクに格納されています。テキストモードでは，入力時には Turbo C は復帰・改行（CR/LF）のペアを改行1文字に変換し，出力時には改行を CR/LF ペアに変換します。

バイナリストリームはテキストストリームよりもずっと単純です。上のような変換はいっさい行なわれず，すべての文字を変換せずに読み書きします。

テキストストリームに関して，このような変換が行われることを理解して使っている限りは，ファイルをテキストモードとバイナリモードのどちらでアクセスしても何の問題もありません。ただし，Turbo C はファイルがどのモードで作成されたか，あるいは最後にどのモードでアクセスされたかを記憶しているわけではありません。

ストリームをオープンする際に変換モードを指定しなかった場合には，グローバル変数 ***_fmode*** で指定されるデフォルトの変換モードでオープンされます。***_fmode*** のデフォルトはテキストモードです。

ストリームのバッファリング

ファイルに結びつけられたストリームは，一般にバッファリングされます。これによって，**getc** や **putc** などの1文字単位の入出力が非常に高速になります。また，**setvbuf** や **setbuf** を呼び出すことによって，自分で指定したバッファを使ったり，バッファの大きさを変えたり，そのストリームがバッファを使わないように指定することもできます。

バッファがいっぱいになったとき，ストリームがクローズされたとき，あるいはプログラムが正常終了したときには，バッファは自動的にフラッシュされます（バッファの内容を対応するストリームに書き出して，バッファをクリアすることをフラッシュといいます）。**fflush** や **fflushall** を使って，任意にバッファをフラッシュすることもできます。

通常，ストリームはシーケンシャルなデータの入出力に使用されます。入出力はその時点のファイル位置に対して行われます。データの読み書きを行なうと，ファイル位置はアクセスしたデータのすぐ後ろに移動します。ディスクファイルに接続されたストリームは，ランダムアクセスすることも可能です。まず **fseek** でファイル位置を決め，その位置（のすぐ後ろ）のデータに対して読み込みや書き出しを行ないます。

1つのストリームに対して，読み込みと書き出しの両方の処理を行なう場合には，これを勝手に混在させることはできません。読み込みと書き出しの間に，そのストリームのバッファをフラッシュする必要があります。**fflush**，**fflushall** あるいは **fseek** を呼び出すと，バッファはクリアされ，読み込み/書き出しの切り換えができるようになります。また移植性を高めるためには，バッファのない場合でもデータをフラッシュしておくべきです。システムによっては，バッファがない場合でも，読み込みと書き出しの混在に制限があるか

もしれません。

定義済みのストリーム

fopen の呼び出しで作成されるストリーム以外に，プログラムの実行中はいつでも使用できるストリームが5つ定義されています。

名前	入出力	モード	ストリーム
stdin	入力	テキスト	標準入力
stdout	出力	テキスト	標準出力
stderr	出力	テキスト	標準エラー出力
stdaux	両方	バイナリ	外部入出力
stdprn	出力	バイナリ	プリンタ出力

stdaux と *stdprn* は MS-DOS に特有のもので，他のシステムへの移植性はありません。

stdin と *stdout* は，MS-DOS によるリダイレクションが可能ですが，残りのストリームは特定のデバイスに接続されています。*stderr* はコンソール（CON:），*stdprn* はプリンタ（PRN:），*stdaux* は補助入出力ポートです。

補助入出力ポートはハードウェアの設定によって異なりますが，通常は COM1:になっています。MS-DOS のコマンドラインでの入出力のリダイレクションについては，MS-DOS のマニュアルを参照してください。リダイレクションをしないときには，*stdin* と *stdout* はコンソール（CON:デバイス）に接続されています。また，リダイレクトされていなければ，*stdin* は行単位のバッファリングが行われますが，*stdout* はバッファリングなしです。他の定義済みストリームはバッファリングされません。

定義済みストリームをデフォルト以外のモードで処理する場合（たとえば *stdprn* をテキストモードで処理する場合）には，**setmode** を使ってください。定義済みストリーム名は定数であり，値を代入することはできません。定義済みストリームに別のファイルやデバイスを結びつけたいときには **freopen** を使ってください。

Cにおけるプログラミングスタイル：“モダン”vs.“古典”

現在，Cのプログラミングに関して，Cをより使いやすくするための構文を言語体系に組み入れて，Cを改革していこうとする動きがあります。こうした改革の多くは，Cプログラミングにおける古典的な伝統や手法には反するものです。改革の大部分は，ANSIのC言語標準化委員会が提案している拡張案で実現されています。この節では，C言語のプログラミングが過去にはどのように行なわれてきたのか，そして新しい標準はよりよいCプログラムを書く上でどんな手助けとなってくれるのかを説明していきます。

もちろん，Turbo Cは古典的なプログラミングスタイルもモダンスタイルも両方サポートしています。

関数プロトタイプの使用と完全な関数の定義

Cのプログラミングの古典的なスタイルでは，関数の宣言は，単に関数の名前と返される値のデータ型を指定することで行ないます。

たとえば，関数*swap*は次のように宣言されます。

```
int  swap();
```

ここでは，引数に関する情報としては，その個数も型も与えられていません。この関数の定義は，古典スタイルでは次のようになります。

```
int  swap(a,b)
int  *a,*b;
{
     /* 関数のボディ */
}
```

このスタイルでは，エラーチェックはほとんど行なわれないことになり，難解で追跡しにくいバグを引き起こしてしまうことがあります。

これを避けるため，モダンスタイルでは，関数の宣言には関数プロトタイプを，関数の定義には引数リストを，それぞれ使用することができます。

*swap*は，関数プロトタイプを用いると，次のように宣言されます。

```
int  swap(int *a, int *b);
```

これで，エラーチェックに必要な情報が用意されるので，プログラムをコンパイルする

ときに，すべての *swap* の呼び出しにおいて完全なエラーチェックが行なわれることになります。関数を定義するときにも同様な書式を使うことができます。

```
int  swap(int *a, int *b)
{
     /* 関数のボディ  */
}
```

モダンスタイルでは，たとえ関数プロトタイプを使用していなくても，より多くのエラーチェックが行なわれます。さらにプロトタイプを使用していれば，コンパイラが，宣言と定義が一致しているかどうかを確認することができます。

enum 定義の使用

古典的なＣのプログラムでは，値のリストは次のように#define 指令で定義されます。

```
#define sun     0
#define mon     1
#define tues    2
#define wed     3
#define thur    4
#define fri     5
#define sat     6
```

しかしモダンスタイルでは，enum というキーワードを使って，列挙型として宣言することができます。上の値のリストを，enum を使って書きかえると次のようになります。

```
typedef enum { sun, mon, tues, wed, thur, fri, sat } days;
```

これは，古典的な方法と同じ効果を持っており，*sun* = 0 ～ *sat* = 6の値をセットしています。ただし，# define を書き並べた場合とくらべると，情報が見えにくくなり，また抽象的になります。さらに，変数を *days* 型として宣言することができます。

typedef の使用

古典スタイルのＣでは，ユーザ定義のデータ型には名前はつけられませんでした。その例外として構造体と共用体がありますが，これらについても宣言の前にはキーワード struct あるいは union をつけなければなりませんでした。

モダンスタイルのＣでは，typedef 命令を使用する際には，enum とは別のレベルで情報の隠蔽が行なわれます。typedef を使うと，（struct と enum も含めて）あるデータ型とユ

ーザが定義する名前を結びつけて，その型の変数を宣言することができます。

以下は，こうした型の定義と変数の宣言の例です。

```
typedef int  *intptr;
typedef char  namestr[30];
typedef enum  { male, female, unknown } sex;
typedef struct {
    namestr  last, first;
    char     ssn[9];
    sex      gender;
    short    age;
    float    gpa;
} student;
typedef student class[100];

class     hist104, ps102;
student   valedictorian;
intptr    iptr;
```

typedefによって，プログラムはより読みやすくなります。また，型を変更する場合にも，実際に型が定義されている箇所を変更すれば，変数が宣言されている箇所をいちいち変更しなくても，その変更はプログラム全体に及びます。

void 型関数の宣言

C 言語のオリジナルの定義では，すべての関数はなんらかの型の値を返していました。型が宣言されていない場合，その関数は int 型を返すとみなされました。また，“汎用の”(型のない) ポインタを返す関数は，通常は char へのポインタを返すと宣言されていました。とにかくなんらかの型の値を返さなければならなかったからです。

現在では，void という標準の型が用意されています。これは一種のヌル(null)型と考えることができます。値を返さない関数は，すべて void 型として明確に宣言すべきです。実行時にメモリの割り当てを行なう関数 (**malloc** など) の多くは，void * という型で宣言されています。これは，型のないポインタを返すという意味であり，(Turbo C では)型を持たないポインタには，型キャストなしでどんな型のポインタへも代入することができます (ただし，プログラムの可搬性を高めるためには，型キャストすることをお薦めします)。

拡張案の利用

C言語は，いくつかの比較的小さな拡張によって，プログラムの可読性が高められ，アナクロニズム（時代遅れの点）が改められ，またユーザの便宜がはかられました。そうした点について少し触れておきます。

文字列リテラル

古典スタイルのCでは，プログラムの中で長い文字列を用いるためには，継続文字を使うか，なんらかの連結の操作を行なわなければなりませんでした。

モダンスタイルでは，次のようにして，長いリテラル（定数）を何行かにわけて記述することができます。

```
main()
{
    char  *msg
    msg = "Four score and seven years ago, our fathers"
          " brought forth upon¥nthis continent a new"
          " nation, dedicated to the ideal that all"
          " men¥nare created equal";

    printf("%s",msg);
}
```

16進文字定数

古典スタイルでは，特定のASCIIコードを表わすエスケープシーケンスは，すべて8進表記（基数が8）で行なわれていました。これは，Cがもともと2進数を8進形式で表現するマシンの上で開発されていたことに起因します。

現在，ほとんどのコンピュータでは，2進数を16進表記（基数が16）で表現しています。このためモダンスタイルでは，文字定数を16進数で宣言することが許されるようになっています。文字定数の一般的な書式は，'¥xDD'です。*DD*は1桁または2桁の16進数（0～9, A～F）を表わしています。このようなエスケープシーケンスは，ch = '¥x20'のようにしてchar型の変数に直接代入することができ，また文字列に組み込むこともできます。

signed型（符号つき）

古典的なCでは，整数を基本にした型はすべて符号つきとみなしていたので，型修飾子unsignedによって，符号なしの型を指定するようになっていました。デフォルトでは，char型は符号つきとみなされたので，値の範囲は-128～127でした。

しかし，現在のマイクロコンピュータでは，char型は符号なしとして考えられることのほうが多く，Turbo Cでも符号なしをデフォルトとするコンパイルオプションが用意されています。符号なしをデフォルトとした場合でも，signed charと宣言すれば符号つきを指定することができます。つまりモダンスタイルでは，符号つきを指定するための型修飾子signedを使うことができるわけです（もちろんunsignedも使えます）。

C プログラミングの落し穴

Cの初心者が往々にして犯しがちなエラーがたくさんあります。そうした落し穴をいくつか紹介し，その対策についてお話していきましょう。

文字列中のパス名

ご存じにように，MS-DOS では¥記号はディレクトリを表わしています。しかしCの文字列の中では，¥はエスケープ文字として扱われます。Cの文字列の中でパス名を使おうとすると，この点が問題になります。

たとえば，次のような文を書いて，

```
file = fopen("c:¥new¥tools.dat","r");
```

ドライブCのディレクトリ NEW にある TOOLS.DAT というファイルをオープンしようとしてもうまくいきません。これでは，¥n は改行文字 (LF)，¥t はタブを表わすエスケープシーケンスとみなされてしまいます。

つまり，指定されたファイル名には，改行とタブが含まれてしまうわけです。ファイル名には改行やタブがあってはならないので，MS-DOS はファイル名が不適切であると判断します。この場合，ファイル名を示す適切な文字列は次のようになります。

```
"c:¥¥new¥¥tools.dat"
```

ポインタの使用法の間違い

初心者にとっては，ポインタが一番の強敵でしょう。どういう場合にポインタを使用すべきか，使用すべきでないか。どういう場合に間接演算子 (*) を使い，アドレス of 演算子(&)を用いるのか。どうすれば，ポインタの誤用によるプログラムの暴走を防げるのでしょうか?

初期化していないポインタを使用してしまう

深刻な間違いの1つに，ポインタにアドレスを代入する前に，そのポインタが指しているアドレスに値を入れてしまうことがあります。

次のプログラムを見てください。

```
main()
{
    int *iptr;

    *iptr = 421;
    printf("*iptr = %d¥n",*iptr);
}
```

この落し穴は非常に危険です。なぜなら，このような誤りがあってもプログラムは正しく動いてしまう場合がよくあるからです。この例では，*iptr* というポインタは，どこか"見当のつかない"アドレスを指しており，そこに421という値が格納されています。このプログラムは小さいものなので，何かが破壊される可能性もごくわずかですが，プログラムが大きくなると，その可能性は格段に増大します。*iptr* がたまたま指しているアドレスに他の情報が格納されているというのは，おおいにあり得ることです。また，タイニィモデルを使用しているとすれば，コードセグメントとデータセグメントは同じ領域を占めているので，機械語コード自体を破壊してしまう危険を冒すことになります。**malloc** を使ってポインタが指すべきアドレスにメモリを用意し，そのアドレスをポインタの中に入れなければならない，ということを憶えておいてください。

文字列

文字列を宣言する2つの方法（char 型へのポインタ，char 型の配列）を思い出したかもしれません。2つの方法はほとんど同じですが，1つだけ重要な違いがありました。この違いとは，char 型ポインタを使用する場合は，文字列を格納する領域はメモリ上に割り当てられないけれども，配列を用いた場合には自動的に割り当てられる，ということでした。

この違いをよく理解していないと，2種類のエラーを引き起こすことがあります。次のプログラムを考えてみましょう。

```
main()
{
    char  *name;
    char  msg[10];

    printf("What is your name? ");
    scanf("%s",name);
    msg = "Hello, ";
    printf("%s%s",msg,name);
}
```

一見，なかなか良さそうなプログラムです。ちょっと不格好かもしれませんが許せる程度でしょう。

しかし，このプログラムには間違いが2つあります。1つ目は，次の文のところです。

```
scanf("%s",name);
```

文法上は正しい文です。*name* は char 型へのポインタなので，アドレス of 演算子（&）をつける必要もありません。

しかしこのプログラムでは，*name* のためのメモリを割り当てていません。したがって，キーボードから入力された名前は，*name* がたまたま指しているどこか“見当のつかない”アドレスに格納されることになります。この文に対してコンパイラは，警告を発しますが（possible use of 'name' before difinition――*name*[のアドレス]が定義される前に使用されている），エラーにはなりません。

2番目の問題は確実にエラーになります。問題は次の文にあります。

```
msg = "Hello, ";
```

コンパイラは，このプログラムは配列名 msg を，文字列定数（"Hello, "）のアドレスに変更しようとしていると考えます。しかし，配列名（つまり配列の先頭アドレス）は定数なので，これを変更することはできません（7は定数なので 7 = i などの代入ができないのと同じです）。コンパイラは Lvalue required というエラーメッセージを出力します。

こうしたエラーはどうすれば解決できるでしょうか。一番簡単な方法は，*name* と *msg* の宣言方法を入れ換えることです。

```
main()
{
    char  name[10];
    char  *msg;

    printf("What is your name? ");
    scanf("%s",name);
    msg = "Hello, ";
    printf("%s%s",msg,name);
}
```

これで完璧です。変数 *name* には，入力された名前を格納する領域を確保していますし，ポインタ *msg* には "Hello," のアドレスを代入することができます。

もし，どうしても宣言の方法を変えたくないとしたら，次のように変更する必要があります。

```
#include <alloc.h>
#include <string.h>

main()
{
    char  *name;
    char  msg[10];

    name = (char *) malloc(10);
    printf("What is your name? ");
    scanf("%s",name);
    strcpy(msg,"Hello, ");
    printf("%s%s",msg,name);
}
```

malloc 関数を呼び出して10バイトのメモリを確保し，そのメモリのアドレスを *name* に代入しています。これで最初の問題が解決されます。**strcpy** 関数は，"Hello,"という文字列定数を配列 *msg* へ1文字ずつコピーします。これで2番目の問題も解決されるわけです。

代入（=）と等号（==）の混同

PascalやBASICでは，式の両辺が等しいかどうかの比較は，if (a = b)という式で行ないます。Cでも，この式は文法上は正しいものですが，意味はまったく異なります。

次のコードを見てください。

```
if (a = b)
    puts("Equal");
else
    puts("Not equal");
```

PascalやBASICのプログラマなら，これは「*a*と*b*が等しければEqualを，等しくなければNot equalを表示する」ことを期待するでしょう。しかしそうではありません。Cでは，a = bという式は，「*a*に*b*の値を代入する」という意味を持ち，この式全体の値は*b*の値になります。したがって上のコードは，「*a*に*b*の値を代入し，*b*の値がゼロ以外ならEqualを，そうでなければNot Equalを表示する」ことになります。

期待どおりの動作をさせるには，次のようにします。

```
if (a == b)
    puts("Equal");
else
    puts("Not equal");
```

switch文にbreakを書き忘れる

breakは，switch文の中で，1つのケースの処理を終了させるために使うということを憶えているでしょう。どうかずっと憶えていてください。ケースの処理の終わりにbreakを書き忘れると，それ以降のケースも続いて実行されてしまいます。

配列の添字

配列は[0]から始まります。[1]からではありません。次のような間違いを起こしがちです。

```
main()
{
    int list[100],i;

    for (i = 1; i <= 100; i++)
        list[i] = i*i;
}
```

このプログラムでは，*list* の最初の記憶場所(*list*[0])が初期化されません。また，list という配列には存在しない記憶場所(*list*[100])に値を格納しており，他のデータの上に重ね書きをしているかもしれません。

正しいコードは，次のようになります。

```
main()
{
    int list[100],i;

    for (i = 0; i < 100; i++)
        list[i] = i*i;
}
```

アドレス渡しを間違える

次のプログラムを見て，どこがいけないのか探し出してください。

```
main()
{
    int a,b,sum;

    printf("Enter two values: ");
    scanf("%d %d",a,b);
    sum = a + b;
    printf("The sum is %d¥n",sum);
}
```

ギブアップですか。エラーはscanf("% d % d",a,b);という文にあります。**scanf**には，値ではなく，アドレスを渡さなければならないことを憶えていたでしょうか。これは，ポインタを引数にとる関数すべてに当てはまります。このプログラムは（エラーなしで）コンパイルが通り，また実行できるでしょう。**scanf**は，*a*と*b*にどんなおかしな値が入っていてもそれをアドレスと考えて，入力された値をそのアドレスに格納してしまうからです。

正しい文は，scanf("% d % d",&a,&b);となります。こうすれば，*a*と*b*のアドレスが**scanf**に渡され，入力した値は2つの変数の中に正しく格納されます。同じような落し穴は，今までに作成した関数にもあり得ます。ポインタに関する節で定義した*swap*関数を思い出してください。

*swap*を，次のように呼び出したらどうなるでしょうか。

```
main()
{
    int i,j;

    i = 421;
    j = 53;
    printf("before: i = %4d j = %4d¥n",i,j);
    swap(i,j);
    printf("after: i = %4d j = %4d¥n",i,j);
}
```

変数*i*と*j*は，*swap*を呼び出す前と後で，同じ値を持っていることになります。さらに，421と53という2つのアドレスに格納されている値が交換されてしまうことになり，これは難解で究明しにくい問題を引き起こすでしょう。

どうすれば，これを回避できるでしょうか。「関数プロトタイプを使って，完全な関数定義を行なう」ことです。

実際，この章の「**ポインタと関数**」のところで説明したように，*swap* が（モダンスタイルで）定義されていれば，この **main** に対してコンパイラがエラーを出してくれます。

ところが，*swap* が次のように定義されていると，プログラムは何の問題もなくコンパイルされてしまいます。

```
void  swap(a,b)
int   *a,*b;
{
      ...
}
```

a と *b* の定義をカッコの外に出すだけで，働くはずのエラーチェックが行なわれなくなってしまいます。これが，古典スタイルの関数定義を使用しない最大の理由です。

出航

第6章の初めにも述べたように，わずか2つの章でCを完全に理解してもらうことは不可能です。しかしとにかく出航は完了しました。いまやるべきことは，これまでに出てきたプログラム例を入力し，コンパイルし，実行してみることです。そして一番大切なのは，プログラムに変更を加えて，何が起こるのかを確認してみることです。幸運を祈ります。Bon voyage.

第8章

Turbo C のビデオ関数

Turbo C には，グラフィックス関数の完全なライブラリが含まれており，カラーでもモノクロでも画面上にグラフや図表を描くことができます。

この章では……

まずはじめに，ビデオモードとウィンドウについて概要を説明します。そのあとで，テキストモードとグラフィックスモードそれぞれについて，どのようにプログラミングを行なうのかを見ていきます。

Turbo C の新しいビデオ関数は，Turbo Pascal で提供されている同様なルーチンに基づいています。PC の画面モードの制御や，ウィンドウあるいはビューポートの作成や管理にまだあまり慣れていない方は，この章で説明しているこうした話題に少し時間をさいてください。

ビデオモードについて

画面の操作モードは，プログラムがモード定義関数（**textmode,initgraph,setgraphmode**）の1つを呼び出したときに定義されます。

■**テキストモード**では，画面はセルによって区切られます（横80または40桁，縦25または20行）。各セルごとに属性と文字を持っています。文字はASCII文字(あるいは漢字)で，属性はその文字がどのように表示されるのか(色，その他)を指定します。Turbo Cは，テキスト画面の操作，画面へのテキストの直接書き込み，およびセル属性の制御を行なうための広範なルーチンを提供しています。

■**グラフィックスモード**では，画面はピクセルで構成されます。ピクセルは，画面に表示される1つのドットです。ピクセルの数(解像度)は，PC-9801シリーズでは横640×縦400です。IBM PCの場合には，システムに接続されているアダプタのタイプとそのモードによって異なります。Turbo Cのグラフィックスライブラリを使って，画面上にグラフィックスを表示することができます。線および図形の描画，閉じた領域のパターンによるフィル（塗りつぶし），またピクセルごとの色の制御などが行なえます。

注意：IBM PCには数種類のビデオアダプタが装着可能です。最も基本的なテキストのみのモノクロディスプレイアダプタ（MDA），グラフィックスの表示が可能なカラーグラフィックスアダプタ（CGA）や拡張グラフィックスアダプタ（EGA），あるいはHerculesモノクログラフィックスアダプタなどがあります。各アダプタは，それぞれさまざまなモードで操作することができます。モードは，画面の幅（40桁または80桁，テキストのみ）や，解像度（グラフィックスのみ），およびディスプレイのタイプ（カラーあるいは白黒）などを指定するものです。Turbo Cのビデオ関数は，(特にグラフィックスにおいては)こうしたアダプタおよびモードのほとんどをサポートしています。

ウィンドウとビューポートについて

ウィンドウの操作

ウィンドウは，テキストモード時のビデオ画面上に定義された長方形の領域です（グラフィックスモード時のものはビューポートと呼びます)。プログラムが画面になにかを書く場合，その出力はアクティブウィンドウに限定され，画面の残りの部分（ウィンドウの外側）はまったく影響を受けません。

デフォルトのウィンドウは画面全体になっています。プログラムは，このデフォルトの全画面からなるテキストウィンドウを，(関数 **window** を呼び出すことによって)小さなテキストウィンドウに変更することができます。この関数は，ウィンドウの位置を画面座標で指定します。

ビューポート

グラフィックスモードでも，画面上に同様な長方形の領域を定義することができます。これが**ビューポート**です。グラフィックスプログラムが図形の出力などを行なう場合，ビューポートは出力の対象となる仮想画面として機能し，画面の残りの部分（ビューポートの外側）にはいっさい影響を与えません。ビューポートは，画面座標に合わせて **setviewport** 関数を呼び出すことによって定義します。

座標

上に示したようなウィンドウ定義あるいはビューポート定義の関数を除いて，テキストモードおよびグラフィックスモード関数に与える座標は，すべてウィンドウ相対またはビュポート相対で指定します。テキストモードのウィンドウでは左上隅が原点で，(1,1) によって参照されます。グラフィックスモードでは，ビューポート座標の原点は (0,0) です。

テキストモードでのプログラミング

この節では，テキストモードで使用する関数の簡単な説明を行なっています。各関数の詳細についてはリファレンスガイドの第2章を参照してください。

Turbo C の直接コンソール I/O パッケージ（**cprintf**，**cputs** など）はより高いパフォーマンスを提供するために大幅に強化されており，効率的なテキスト出力，ウィンドウの管理，カーソルの位置づけ，属性の制御などが行なえるようになっています。これらの関数は，すべて Turbo C 標準ライブラリの一部であり，ヘッダファイル conio.h の中でプロトタイプ宣言されています。

コンソール I/O 関数

Turbo C のテキストモード関数は，8つのテキストビデオモード（IBM PC では5つ）のいずれにおいても動作します。テキストモードは，**textmode** を呼び出して指定します。この関数の使い方はもう少し後で説明しますが，詳細についてはリファレンスガイドの第2章を見てください。

これらのテキストモード関数は，次のように4つのグループに分類されます。

- ■テキストの出力と操作
- ■ウィンドウとモードの制御
- ■属性の制御
- ■状態の問い合わせ

コンソール I/O 関数は標準入出力関数とは異なり，テキストの出力を MS-DOS を介さず直接テキスト VRAM 上に書き込みます（IBM PC の場合は，ビデオ RAM に直接書き込むか，BIOS を通じて表示するかを指定することができます）。そのため表示速度は非常に高速ですが，これらの関数を使ったプログラムは，PC-9801シリーズ（あるいは IBM PC）以外の機種では動かないことになります。

注意：IBM PC では，**textmode** 関数によって選択可能なテキストモードは，システムが備えているアダプタの種類によって異なります。

以下では，これらのテキストモード関数の4つのグループについて説明していきます。なお属性の制御については，PC-9801シリーズとIBM PCではハードウェアの特性の違いから大きく異なっているため，節をわけて説明しています。

テキストの出力と操作

テキストの出力と操作を行なう関数には以下のものがあります。

■テキストの読み書き:

cprintf	書式つき出力を画面に送る
cputs	文字列を画面に送る
putch	文字1個を画面に送る
getche	文字を読み，その文字を画面にエコーバックする

■画面上のテキスト（およびカーソル）の操作：

clrscr	テキストウィンドウのクリア
clreol	カーソルから行末までのクリア
delline	カーソルがある行の削除
gotoxy	カーソルの位置づけ
insline	カーソルがある行の下に空白行を1行挿入する
movetext	画面上のある領域のテキストを他の場所へコピーする

■テキストブロックをメモリからロードしたり，メモリにセーブしたりする：

gettext	画面上のある領域のテキストをメモリにコピーする
puttext	メモリから画面上のある領域にテキストをコピーする

画面出力を行なうプログラムは，テキストウィンドウが画面全体というデフォルトの状態でスタートします。したがって，前もってモードの設定を行なわなくても，すぐにテキストの読み書きや操作することができます。直接コンソール出力関数**cprintf**，**cputs**，**putch**を用いて画面にテキストを書いたり，**getche**を用いてエコーつきの入力をしたりすることができます。テキストはウィンドウの中では“つながって”います。つまり，テキストがウィンドウの右端を越えるまでに広がってしまうような場合，ウィンドウの右端を越える部分は次の行の先頭に移動します。

テキストが画面上に表示されているときには，**clrscr** でアクティブウィンドウ（現在有効なウィンドウ）の内容を消去する，**clreol** で行の一部を消す，**delline** で行全体を削除する，**insline** で空白行1行を挿入することができます。後の3つの関数は，カーソルの位置に関連して操作を行なようになっており，カーソル位置は **gotoxy** によって指定の位置に動かすことができます。また，長方形のテキストのブロック全体を，**movetext** でウィンドウの中のある場所から他の場所へコピーすることもできます。

gettext で画面上の長方形のテキストブロックをメモリに格納したり，**puttext** でそのテキストを画面上の任意の場所に復元することもできます。

ウィンドウとモードの制御

ウィンドウおよびモードの制御を行なう関数には以下のものがあります。

textmode	画面をテキストモードに設定する
window	ウィンドウを定義する
textbank	画面ページの切り換えを行なう（PC-9801のみ）

textmode によって，いくつかあるテキストビデオモードのうちのどれかに画面を設定することができます。テキストモードをセットすると，画面は全画面のテキストウィンドウとして初期化され，指定されたモードで，表示されていたイメージおよびテキストはすべてクリアされます。

画面がテキストモードの場合，全画面のウィンドウに出力することもできますが，画面の一部（ウィンドウ）だけに出力を限定することもできます。テキストウィンドウを作成するためには，画面上のどの領域にウィンドウを設定するかを指定して，**window** を呼び出します。

textbank を使うと，表示する画面ページを0ページと1ページの間で切り換えることができます。各画面ページにおいては，テキストモードは共通ですが，ウィンドウの大きさ，文字の属性，カーソルの属性などはそれぞれのページに独立して制御することができます。なお，テキストモードが80桁×25行の場合には，0ページのカーソル位置は，MS-DOS が保持しているカーソル位置に一致します。テキストモードが20行または40桁のモードのときには，カーソル位置は MS-DOS とは独立に管理されます。

属性制御（PC-9801）

以下は，PC-9801のテキストモードの属性を制御する関数です。

■文字の表示属性の設定:

textattr	文字の表示属性を設定する
textcolor	文字の色（属性）を設定する
textblink ＊	文字の点滅（属性）を設定する
textreverse ＊	文字の反転（属性）を設定する
textunder ＊	文字の下線（属性）を設定する
textvertical ＊	文字の縦線（属性）を設定する
normvideo	元の表示属性を復元する

■カーソル属性の設定:

textcursor ＊	カーソルの表示属性を設定する

注：＊ PC-9801用にのみ用意されている関数です。

属性制御関数を使うと，文字やカーソルの現在の属性を変更することができます。

文字の属性は8ビット値で表現され，最下位から順に，(0)隠し属性，(1)点滅，(2)反転，(3)下線の有無，(4)縦線の有無またはグラフィック文字の表示のオン/オフ(画面モードによって意味が変わります)の意味を持っており，上位の3ビット(5～7)は文字の色を表わしています。

属性設定に続いて出力されるテキストは，現在の属性によって表示されます。上に示した各属性は，それぞれ別個に設定することも，**textattr** 関数の呼び出しだけでまとめて設定することもできます。

textcursor を使うと，カーソルの表示/非表示，点滅/静止を指定することができます。

なお，IBM PC 版との互換性維持のために，**highvideo**，**lowvideo**，**textbackground** の3つ関数は，実際にはなんの動作もしないダミー関数として定義されています。

属性制御（IBM PC）

以下は，IBM PC におけるテキストモードの属性を制御する関数です。

■フォアグラウンドとバックグラウンドの設定：

textcolor	フォアグラウンド色（属性）を設定する
textbackground	バックグラウンド色（属性）を設定する
textattr	フォア/バックグラウンド色（属性）を同時に設定する

■輝度（intensity）**の切り換え**:

highvideo	テキストを高輝度に設定する
lowvideo	テキストを低輝度に設定する
normvideo	テキストを元の輝度に設定する

属性制御関数は，現在の属性を設定します。属性は8ビット値で表現され，下位4ビットがフォアグラウンド色，次の3ビットがバックグラウンド色，最上位の1ビットがブリンク（点滅）の有無を示しています。

属性設定に続いて出力されるテキストは，現在の属性によって表示されます。属性制御関数を使うと，バックグラウンドとフォアグラウンドの（文字の）色を別々に設定することも（**textcolor** と **textbackground** を使う），**textattr** を呼び出すだけで色の指定を組み合わせて設定することもできます。さらに，それ以降の（フォアグラウンドの）文字をブリンク（点滅）させるかどうかも指定することができます。ほとんどのカラーモニタでは，その通りの色で表示されます。カラーでないモニタの場合は，属性のいくつか（あるいはすべて）を，階調表示や他の効果（太字，下線，反転など）によって代替することがあります。

lowvideo を使うと，フォアグラウンドの高輝度色を低輝度色に切り換える（文字の高輝度ビットをオフにする）ようシステムに指示することができます。また，**highvideo** を使って，低輝度色を高輝度色に切り換える（文字の高輝度ビットをオンにする）こともできます。文字の輝度をいろいろと切り換えるときには，**normvideo** によって元の値を復元することができます。

状態の問い合わせ

以下は状態問い合せ関数の要約です。

gettextinfo	現在のテキストウィンドウに関する情報を text _info 構造体に代入する
wherex	カーソル位置のセルの x 座標を与える
wherey	カーソル位置のセルの y 座標を与える

Turbo C のコンソール入出力関数には，状態の問い合せのために用意されたものがいくつかあります。これらの関数によって，テキストモードウィンドウやウィンドウ内でのカーソルの現在位置などの情報を取り出すことができます。

gettextinfo は，**text _info** 構造体（conio.h の中で定義されている）の中に，テキストウィンドウについての次のような情報を代入します。

■現在のビデオモード
■絶対画面座標でのウィンドウの位置
■ウィンドウの大きさ
■現在のフォアグラウンドとバックグラウンドの色
■カーソルの現在の位置

これらの情報を全部必要とすることはあまりなく，単にカーソルの位置（ウィンドウ相対の）を知るには，**wherex** と **wherey** を使う方が便利です。

テキストウィンドウ

デフォルトのテキストウィンドウは全画面で，**window** を呼び出して，これを小さなテキストウィンドウに変更することができます。テキストウィンドウは，20あるいは25行まで（すべてのテキストモードでの画面上の最大行数），40あるいは80桁まで（テキストモードに依存します）可能です（IBM PC では，25行モードしか使用できません）。

Turbo C のテキストウィンドウ座標の原点は，ウィンドウの左上隅です。ウィンドウの

左上隅の座標は（1,1）です。したがって，全画面が80桁の幅のテキストウィンドウの右下隅の座標は（80,25）となります。

例

80桁テキストモードで，ウィンドウを作成する場合を考えてみましょう。ウィンドウの左上隅が画面座標（10,8），ウィンドウの右下隅が画面座標（50,21）となるようにするには，**window** を次のように呼び出します。

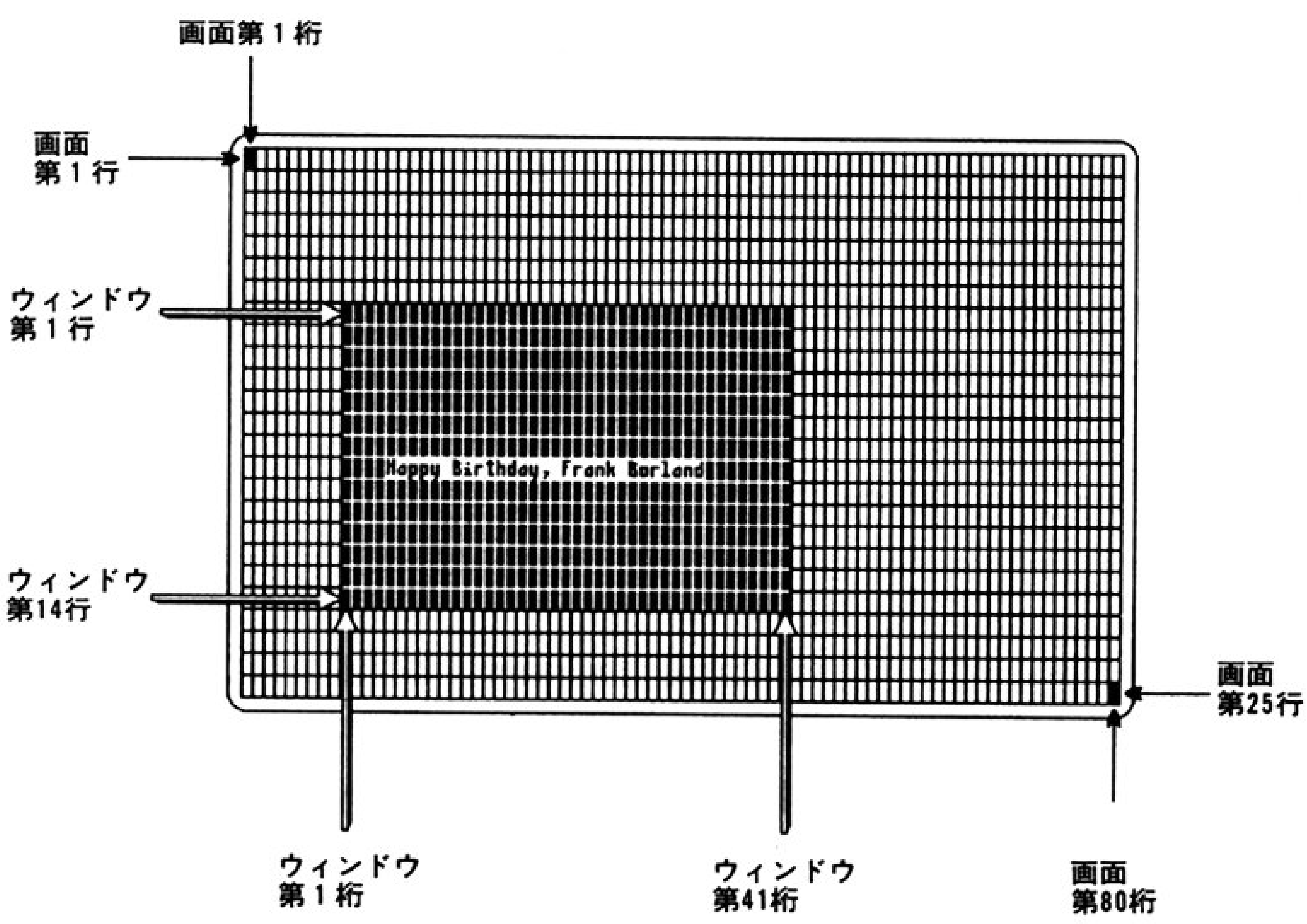
```
window(10, 8, 50, 21);
```

これでテキストモードウィンドウを作成したことになるので，次にウィンドウ位置(5,8)にカーソルを動かして，あるテキストをそこに書きたければ，**gotoxy** と **cputs** を使って，次のようにすればよいことになります。

```
gotoxy(5,8);
cputs("Happy Birthday, Frank Borland");
```

このときの画面は**図 8.1** のようになります。

図 8.1　80×25 のテキストモードでのウィンドウ



text _modes 型

textmode 関数を呼び出すことによって，モニタを8つあるテキストモードの中の1つに対応するようにすることができます。conio.h で定義されている enum 型 *text _modes* を使うと，**textmode** への引数 *mode* を直接数値で指定するかわりに，記号名で指定することができます。ただし記号定数を使うときは，ソースコードにインクルード文を置いて，ヘッダファイルを取り込まなければなりません。

```
#include <conio.h>
```

text _modes で定義される数値，記号は次に示すとおりです。

シンボリック定数	値	画面のテキストモード
■ **PC-9801**：		
LASTMODE	-1	直前のテキストモード
VL8025	0	80桁×25行，縦罫線が有効
VL8020	1	80桁×20行，縦罫線が有効
VL4025	2	40桁×25行，縦罫線が有効
VL4020	3	40桁×20行，縦罫線が有効
BG8025	4	80桁×25行，グラフィック文字が表示可能
BG8020	5	80桁×20行，グラフィック文字が表示可能
BG4025	6	40桁×25行，グラフィック文字が表示可能
BG4020	7	40桁×20行，グラフィック文字が表示可能
■ **IBM PC**：		
LASTMODE	-1	直前のテキストモード
BW40	0	白黒，40桁
C40	1	16色，40桁
BW80	2	白黒，80桁
C80	3	16色，80桁
MONO	4	モノクロ，80桁

たとえば，**textmode** を次のように呼ぶと，カラーモニタは右側で示されるオペレーティングモードになります。

呼び出し	オペレーティングモード
■ PC-9801：	
textmode(2)	40桁，25行，縦罫線が有効
textmode(BG8020)	80桁，20行，グラフィック文字が表示可能
■ IBM PC：	
textmode(0)	白黒，40桁
textmode(BW80)	白黒，80桁
textmode(C40)	16色，40桁
textmode(3)	16色，80桁

テキストの色

あるセルの属性がどのように割り当てられているかといった詳細については，リファレンスガイド第2章の **textattr** の解説を参照してください。

PC-9801では，カラーモニタは8色まで表示できます。モノクロディスプレイでは色の違いは階調で表現されます。また，色の他に反転や下線などの属性を付加することもできます。

IBM PC では，セル中の文字の色がフォアグラウンド，セルの文字以外の範囲がバックグラウンドで，それぞれ別々に指定することができます。カラービデオアダプタにカラーモニタを装着していれば16色まで表示可能です。モノクロモニタでは，これらの色はさまざまな表示属性（輝度，下線，反転など）によって表現されます。

インクルードファイル conio.h の中には，いろいろな色や他の属性に対するシンボリック定数が定義されています。この記号名を使うときは，やはり conio.h をインクルードしてください。

次の 2 つの表は，シンボリック定数とそれが表わす値，およびその意味を示しています。

表 8.1　PC-9801 のテキストモードで使用するシンボリック定数

シンボリック定数	値	意味
T _BLACK	0x00	黒
T _BLUE	0x20	青
T _GREEN	0x80	緑
T _CYAN	0xA0	水色
T _RED	0x40	赤
T _MAGENTA	0x60	紫
T _BROWN	0xC0	黄色
T _LIGHTGRAY	0xE0	白
T _DARKGRAY	0x00	黒
T _LIGHTBLUE	0x20	青
T _LIGHTGREEN	0x80	緑
T _LIGHTCYAN	0xA0	水色
T _LIGHTRED	0x40	赤
T _LIGHTMAGENTA	0x60	紫
YELLOW	0xC0	黄色
WHITE	0xE0	白
VERTICALLINE	0x10	縦線の添付
NOVERTICALLINE	0x00	
UNDERLINE	0x08	下線の添付
NOUNDERLINE	0x00	
REVERSE	0x04	反転表示
NOREVERSE	0x00	
BLINK	0x02	点滅表示
NOBLINK	0x00	
SECRET	0x00	
NOSECRET	0x01	隠し属性の否定
BASICGRAPH	0x10	グラフィック文字の指示
NOBASICGRAPH	0x00	

表 8.2　IBM PC のテキストモードで使用するシンボリック定数

シンボリック定数	値	意味	フォア/バック
BLACK	0	黒	両方
BLUE	1	青	両方
GREEN	2	緑	両方
CYAN	3	水色	両方
RED	4	赤	両方
MAGENTA	5	紫	両方
BROWN	6	茶色	両方
LIGHTGRAY	7	明るい灰色	両方
DARKGRAY	8	暗い灰色	フォアグラウンドのみ
LIGHTBLUE	9	明るい青	フォアグラウンドのみ
LIGHTGREEN	10	明るい緑	フォアグラウンドのみ
LIGHTCYAN	11	明るい水色	フォアグラウンドのみ
LIGHTRED	12	明るい赤	フォアグラウンドのみ
LIGHTMAGENTA	13	明るい紫色	フォアグラウンドのみ
YELLOW	14	黄色	フォアグラウンドのみ
WHITE	15	白色	フォアグラウンドのみ
BLINK	128	点滅	フォアグラウンドのみ

IBM PC の場合，最初の8色（BLACK～LIGHTGRAY）はフォア/バック両方に指定できますが，その後の8色（DARKGRAY～WHITE）はフォアグラウンド（文字そのもの）にしか指定できません。また，シンボリック定数 BLINK（値128）をフォアグラウンドの指定に加えると，文字を点滅表示させることができます。

ハイパフォーマンスアウトプット:directvideo 変数

Turbo C のコンソール I/O パッケージには ***directvideo*** という変数が含まれています。この変数は，プログラムの行なうコンソール出力が，直接ビデオ RAM に送られる（***directvideo*** = 1）のか，あるいは BIOS コールを通じて出力される（***directvideo*** = 0）のかをコントロールしています。

デフォルト値は ***directvideo*** = 1（コンソール出力は直接ビデオ RAM に送られる）です。一般に，ビデオ RAM へ直接送ることによって，非常に高いパフォーマンスが得られますが，これを行なうには，使用するコンピュータが IBM PC と100%コンパチブルでなければなりません。つまり，ビデオハードウェアが IBM のディスプレイアダプタとまったく同じ機能を有していなければならないわけです。***directvideo*** = 0にセットすると，IBM PC と BIOS レベルでコンパチブルなマシンであれば動作させせることができますが，コンソール出力はやや遅くなります。

注意：PC-9801シリーズ用の Turbo C では，***directvideo*** 変数は互換性のために定義はされていますが，使用されることはまったくありません。PC-9801シリーズでは，コンソール出力関数による出力はすべてテキスト VRAM に直接送られます。

漢字の出力

全角文字の表示は，第1バイトと第2バイトのコードを連続して出力する必要があります。ただし，第2バイト目のコードが不適当なコードであると，その全角文字は空白として表示されます。また，画面の最右端に全角文字を表示しようとした場合は，その文字は次行の先頭から表示されます。

ウィンドウ内での制御コード

テキストウィンドウ内で，コンソール出力関数によって以下の制御コードを出力すると，右に示したような動作をします。

BELL	0x07	ベルを鳴らす
TAB	0x09	タブ（8文字間隔）
BS	0x08	カーソルを1文字戻す
CR	0x0D	カーソルを行の先頭に戻す
LF	0x0A	カーソルを次の行に移動する

グラフィックスモードでのプログラミング

この節では，グラフィックスモードで用いる関数の概要を説明します。各関数の詳細についてはリファレンスガイド第2章の該当する解説を参照してください。

Turbo C では，高レベルの関数（**setviewport,bar3d,drawpoly** など）から，ビット指向の関数（**getimage** や **putimage** など）にいたるまで，70以上のグラフィックス関数を含むライブラリが別個に用意されています。このグラフィックスライブラリは，フィル（塗りつぶし）とドロウ（線の描画）のさまざまなスタイルをサポートしています。また，大きさの変更，位置合わせ，出力方向の指定（水平/垂直）が行なえるテキストフォントを持っています。

こうした関数は，グラフィックスライブラリ GRAPHICS.LIB の中におさめられており，ヘッダファイル GRAPHICS.H の中でプロトタイプ宣言されています。この2つのファイルに加えて，グラフィックスパッケージには，グラフィックスデバイスドライバ(*.BGI ファイル）とストロークフォント（*.CHR ファイル）が含まれています。この2つのファイルについては次の節で説明します。

グラフィックス関数を使用するときは，次のようにします。

- TC.EXE を使っている場合は，**Options/Linker/Graphics library** トグルを On にセットしてください。こうすると，プログラムを Make するときに，リンカが自動的に Turbo C グラフィックスライブラリをリンクしてくれます。
- TCC.EXE を使うときには，コマンドラインで GRAPHICS.LIB を指定する必要があります。たとえば，グラフィックスを使っているプログラム MYPROG.C をコンパイル・リンクするときには，TCC のコマンドラインは次のようになります。

```
tcc myprog.c graphics.lib
```

重要な注意：グラフィックスライブラリファイルは GRAPHICS.LIB ひとつだけで，メモリモデルごとに別々のバージョンが用意されているわけではありません（標準ライブラリはメモリモデルごとに，CS.LIB，CC.LIB，CM.LIB のようにわかれています）。GRAPHICS.LIB に含まれているすべての関数は far 関数で，さらにポインタを引数にとるグラフィックス関数は，far ポインタをとります。これらの関数を正しく機能させるためには，グラフィックスを使うすべてのモジュールで，次のようにヘッダファイル GRAPHICS.H をインクルードしなければなりません。

```
#include <graphics.h>
```

グラフィックスライブラリ関数

Turbo C のグラフィックス関数は次の7つのカテゴリからなっています。

■グラフィックスシステム制御
■ドロウ（draw－図形の描画）とフィル（fill－塗りつぶし）
■画面とビューポートの操作
■テキスト出力
■カラーの制御
■エラー処理
■状態の問い合わせ

PC-9801シリーズ用のグラフィックスパッケージは，オリジナルのIBM PC のものとほぼ同等の機能を実現していますが，ハードウェアの特性上，カラーの制御に関しては両バージョンでは大きな違いがあるため，節をわけて解説しています。

グラフィックスシステムの制御

以下にグラフィックスシステム制御関数の要約を示します。

closegraph	グラフィックスシステムを終了する
detectgraph	ハードウェアを調べて使用するグラフィックスドライバを決定し，推奨するグラフィックスモードを提示する
graphdefaults	グラフィックスシステムのすべての変数をデフォルト値にリセットする
_graphfreemem	グラフィックスメモリの領域を解放する（ユーザ専用のルーチンを定義するためのフック）
_graphgetmem	グラフィックスメモリの領域を割り当てる（ユーザ専用のルーチンを定義するためのフック）
getgraphmode	カレントグラフィックスモードを返す
getmaxmode	与えられたドライバに有効なモードの最大値を返す

getmodename	カレントのグラフィックスモード名を返す
getdrivername	カレントのグラフィックスドライバ名を返す
initgraph	グラフィックスシステムを初期化して，ハードウェアをグラフィックスモードにする
installuserdriver	デフォルトではないデバイスドライバを，BGI デバイスドライバテーブルに組み込む
installuserfont	グラフィックスルーチンが認識していないストロークフォントファイルをロードする
registerbgidriver	リンクされた，あるいはユーザがロードしたドライバファイルをグラフィックスシステムに登録する
registerbgifont	リンクされたストロークフォントコードを登録する
restorecrtmode	**initgraph** を呼び出したときの画面モードに戻す
setgraphbufsize	内部グラフィックスバッファの大きさを決める
setgraphmode	指定されたグラフィックスモードを選択し，画面をクリアして，すべてをデフォルトにセットする

PC-9801シリーズ用の Turbo C のグラフィックスパッケージには，グラフィックスドライバとして次の3つが含まれています。

■ PC98.BGI（PC-9801シリーズすべてで動作します）
■ PC98GRCG.BGI（グラフィックチャージャを備えたマシンで動作します）
■ PC98EGC.BGI（EGC を備えたマシンで動作します）

IBM PC 用のグラフィックスパッケージは，以下のグラフィックスアダプタ（および完全互換アダプタ）をサポートするグラフィックスドライバが含まれています。

■カラーグラフィックスアダプタ（CGA）
■マルチカラーグラフィックスアダプタ（MCGA）
■拡張グラフィックスアダプタ（EGA）
■ビデオグラフィックスアレイ（VGA）
■ Hercules グラフィックスアダプタ
■ AT&T 400ライン・グラフィックスアダプタ
■3270 PC グラフィックスアダプタ
■ IBM 8514グラフィックスアダプタ

グラフィックスシステムをスタートするには，まず最初に **initgraph** を呼び出します。**initgraph** には，特定のグラフィックスドライバおよびモードを使うように指示することができ，また実行時にマシンの装備を検出して，自動的に装備に対応するドライバをロードさせるようにすることもできます。**initgraph** にこのような自動検出を指示すると，**initgraph** は **detectgraph** を呼び出してグラフィックスドライバとモードを選択します。**initgraph** に特定のドライバとモードを使うように指示する場合，それに必要なハードウェアがマシンに装備されていなければなりません。強制的に装備されていないハードウェアを使おうとすると，その結果は予測できないものになります。

なんらかのグラフィックスドライバがロードされると，**getdrivername** 関数を使ってそのドライバの名前を取り出したり，**getmaxmode** によってそのドライバがいくつのモードをサポートしているかを調べたりすることができます。また **getgraphmode** を使えば，現在のグラフィックスモード番号を知ることができます。モード番号がわかれば，**getmodename** によってそのモードの名前を得ることができます。グラフィックスモードは **setgraphmode** を使って切り換えることができ，**restorecrtmode** によってビデオモードを元の状態（グラフィックスを初期化する前の状態）に戻すことができます。

graphdefaults は，グラフィックスの状態の設定（ビューポートのサイズ，ドロウカラー，フィルカラーとパターンなど）をデフォルトにリセットします。

installuserdriver と **installuserfont** は，デフォルト以外の新しいデバイスドライバおよびフォントをグラフィックスシステムに追加するための関数です。

グラフィックスを使い終えたら，最後に **closegraph** を呼び出してグラフィックスシステムを終了させます。**closegraph** は，ドライバが占めていたメモリを解放し，画面を（**restorecrtmode** を通じて）元のビデオモードに復帰させます。

注意：PC-9801では，Turbo C 1.5で使用されていた **setnewdriver** も残されています。これは内部では **installuserdriver** を呼び出しており，互換性のためにのみ用意されています。なお，バージョン2.0では **installuserdriver**（あるいは **setnewdriver**）を使わなくても，自動検出によって適切なドライバをロードすることができます。このようにして作成した実行可能プログラムは，IBM PC の上でも（適切なドライバがあれば）そのまま動作します。したがって，新しいプログラムでは **setnewdriver** はなるべく使わないようにした方がよいでしょう。

もう少し詳しい解説

前の説明で，**initgraph** がどのように働くのかについてはおおまかに述べました。次に，

initgraph, **_graphgetmem**, **_graphfreemem** の働きを少し詳しく説明します。

通常，**initgraph** ルーチンは，グラフィックスドライバ用にメモリを割り当て，適切な.BGI ファイルをディスクから読み込んで，ドライバをロードします。

このように動的にドライバを組み込む方法の他に，実行可能プログラムの中に，グラフィックスドライバファイルを（場合によっては複数個）直接リンクする方法があります。これには，まず（BGIOBJ ユーティリティを用いて）.BGI ファイルを.OBJ ファイルに変換し，それからグラフィックスドライバを登録するために，**registerbgidriver** の呼び出しをソースコード中に（**initgraph** の呼び出しより前に）置きます。そして，実行可能プログラムを作成するときに，登録されたドライバに対応する.OBJ ファイルをリンクします。

initgraph は（**detectgraph** を通して）使用するグラフィックスドライバを決定した後，そのドライバがメモリ上に登録されているかどうかをチェックします。登録されていれば，**initgraph** はそのドライバをメモリから直接使用し，登録されていなければ，ドライバ用にメモリを割り当ててディスクから.BGI ファイルを読み込みます。

注意：**registerbgidriver** を使うには，少し高度なプログラミングテクニックを必要とするので，経験の浅いプログラマにはこの方法はすすめられません。この関数についてはこの章の他に，付録 D の「**BGIOBJ**」の節でも少し詳しく説明しています。

グラフィックスシステムは実行時に，ドライバ，フォント，および内部バッファ用のメモリを割り当てなければならない場合があります。このような場合，グラフィックスシステムは **_graphgetmem** を呼び出してメモリを割り当て，**_graphfreemem** を呼び出してそのメモリを解放します。これらのルーチンは，デフォルトではそれぞれ単に **malloc** と **free** を呼んでいるだけです。

ユーザが独自の **_graphgetmem** および **_graphfreemem** を定義すれば，このデフォルトを変更することができます。グラフィックスメモリの割り当てをユーザ側で制御することができるわけです。ただし，ユーザ独自のメモリ割り当てルーチンには，同じ名前を使わなくてはなりません。つまり，標準 C ライブラリの中にあるのと同じ名前を使って，デフォルトの解放を無視するわけです。

注意：ユーザ独自の **_graphgetmem** と **_graphfreemem** を定義してプログラムを作成すると，グラフィックスライブラリの中に同名の関数があるため，リンク時に“duplicate symbol”（シンボルの重複）という警告メッセージが出ます。この警告は無視してください。

ドロウとフィル

次に，ドロウ（図形の描画）とフィル（塗りつぶし）関数の要約を説明します。

■ドロウ：

arc	円弧を描く
circle	円を描く
drawpoly	多角形の輪郭を描く
ellipse	楕円の円弧を描く
getarccoords	**arc** あるいは **ellipse** の最後の呼び出し時の座標を返す
getaspectratio	カレントグラフィックスモードのアスペクト比を返す
getlinesettings	カレントラインスタイル，ラインパターン，ライン幅を返す
line	$(x0, y0)$ から $(x1, y1)$ まで直線を描く
linerel	現在位置（CP）から指定の長さ（相対距離）の直線を描く
lineto	現在位置（CP）から (x,y) まで直線を描く
moveto	現在位置（CP）を (x,y) に動かす
moverel	現在位置（CP）を指定の相対距離だけ動かす
rectangle	長方形を描く
setlinestyle	ライン幅とラインスタイルをセットする
setaspectratio	デフォルトのアスペクト比調整因子を変更する

■フィル：

bar	バー（長方形）を描いて内部をフィルする（塗りつぶす）
bar3d	3次元のバーを描いてフィルする
fillellipse	楕円を描いてフィルする
fillpoly	多角形を描いてフィルする
floodhill	閉鎖領域をフィルする
getfillpattern	ユーザ定義のフィルパターンを返す
getfillsetting	カレントフィルパターンとフィルカラーに関する情報を返す
pieslice	パイスライス（扇形）を描いてフィルする

sector	楕円のパイスライス（扇形）を描いてフィルする
setfillpattern	ユーザ定義のフィルパターンを選ぶ
setfillstyle	フィルパターンとフィルカラーをセットする

Turbo Cのドロウおよびフィルの関数を使うと，カラーで直線，弧，円，楕円，長方形，パイスライス，2次元/3次元のバー，多角形，およびこれらを組み合わせて規則的な図形あるいは不規則な図形を描くことができます。閉じた図形（あるいはそのような図形で囲まれた領域）なら，11種類の定義済みパターンあるいはユーザ定義のパターンのうちの1つで塗りつぶしを行なうことができます。また，線の幅とスタイル，現在位置(CP)も自由に変えることができます。

arc，**circle**，**drawpoly**，**ellipse**，**line**，**linerel**，**lineto**，**rectangle** といった関数を使って，線や中空の図形を描きます。この図形は **floodfill** でフィルする（塗りつぶす）ことができます。また，**bar**，**bar3d**，**fillellipse**，**fillpoly**，**pieslice** および **sector** を使えば，ドロウ（線を描く）とフィルの操作を1ステップで行なうことができます。**setlinestyle** を使うと，ライン（フィルする図形の輪郭線も）の幅，およびラインスタイル（実線，点線，ユーザ定義のパターンなど）を選択することができます。**setfillstyle** によって定義済みのフィルパターンから選択することができ，**setfillpattern** を使えばユーザがフィルパターンを定義することができます。現在位置（CP）を指定の場所に動かすには **moveto** を使い，指定の距離だけ動かすには **moverel** を使います。

カレントラインスタイルと幅を知るには，**getlinesettings** を呼び出します。カレントフィルパターンとカラーを知るには，**getfillsettings** を呼び出します。一方，ユーザ定義のフィルパターンを得るには **getfillpattern** を用います。

アスペクト比を得るには **getaspectratio** を用います(アスペクト比は，円を真円として表示するために使用される，CRT 装置のピクセルの縦横の比率です)。画面に表示された円が真円でないときには，**setaspectratio** を使ってアスペクト比を調整することができます。前回描いた円弧あるいは楕円の座標を得るには **getarccoords** を呼びます。

画面とビューポートの操作

次にイメージ操作関数の要約を示します。

■画面操作：

cleardevice	画面をクリアする（アクティブページ）

setactivepage	グラフィックスを出力するアクティブページをセットする
setvisualpage	ビジュアルグラフィックスページ番号をセットする

■ビューポート操作：

clearviewport	カレントビューポートをクリアする
getviewsettings	カレントビューポートに関する情報を返す
setviewport	グラフィックスを出力するカレント出力ビューポートをセットする

■イメージ操作：

getimage	指定領域のビットイメージをメモリに書き込む
imagesize	画面のある長方形領域を退避するのに何バイト必要かを返す
putimage	以前に書き込んでおいたビットイメージを画面上に表示する

■ピクセル操作：

getpixel	(x,y) のピクセルカラーを得る
putpixel	(x,y) にピクセルをプロットする

ドロウやフィル以外にも，グラフィックスライブラリには，画面，ビューポート，イメージ，ピクセルを操作するために関数が用意されています。**cleardevice** を呼び出すと，一度にすべての画面をクリアします。このルーチンは画面を消去して現在位置（CP）をビューポートのホームポジションに移しますが，グラフィックスシステムの他のすべての設定は変更しません（ライン，フィル，およびテキストのスタイル，パレット，ビューポートの設定)。

PC-9801シリーズでは，使用するマシンによって，システムが持つ画面ページバッファの数は1か2になります（最初期型および PC-9801U2が1ページ，それ以外は2ページです)。IBM PC の場合は，アダプタのタイプによって画面ページの数は1～8になります。このバッファはメモリ上の領域で，画面イメージ全体が1ドットごとに格納されています。どの画面ページをアクティブにするか（グラフィックス出力はアクティブページに対して行なわれます)，ビジュアルページをどれにするか (実際の画面に表示されるページ）を指定するには，それぞれ **setactivepage**，**setvisualpage** を使います。

画面をグラフィックスモードにすると，**setviewport** を呼び出すことによって，画面上にビューポート（長方形の仮想画面）を定義することができます。ビューポートの位置は絶対画面座標によって定義し，クリッピングがオン（有効）かオフかも指定します。ビューポートのクリアは **clearviewport** を使って行ないます。カレントビューポートの絶対画面座標とクリッピングの状態を知るには，**getviewsettings** を呼び出します。

画面上のイメージの一部を得るには **getimage** を用います。**imagesize** を呼び出せば，そのイメージをメモリに格納するのに何バイト必要かを計算できます。そして格納したイメージを画面上の好きな場所に復元するには **putimage** を使います。

出力関数（ドロウ，フィル，テキスト）に対して指定する座標は，すべてビューポート相対座標を用います。

個々のピクセルのカラーを操作するには，**getpixel**（指定されたピクセルのカラーを返す）と **putpixel**（指定されたピクセルをドロウカラーで描く）を使います。

グラフィックスモードでのテキスト出力

次にグラフィックスモードでのテキスト出力関数の要約を示します。

gettextsettings	カレントテキストフォント，向き，サイズ，桁揃えしているかを返す
outtext	画面の現在位置（CP）に文字列を書く
outtextxy	画面の指定位置に文字列を書く
registerbgifont	リンクされた，あるいはユーザがロードしたフォントを登録する
settextjustfy	**outtext** と **outtextxy** で使用するテキストの位置合わせの状態値をセットする
settextstyle	カレントテキストフォント，スタイル，文字拡大因子をセットする
setusercharsize	ストロークフォントの幅と高さの比を設定する
textheight	文字列の高さをピクセル単位で返す
textwidth	文字列の幅をピクセル単位で返す

グラフィックスライブラリには，グラフィックスモードでのテキスト出力のために，8×16ビットマップフォントと，いくつかのストロークフォントが含まれています（IBM PC

ではデフォルトのビットマップフォントは8×8です)。

■**ビットマップフォント**の各文字は，ピクセルの行列で定義されています。
■**ストロークフォント**の各文字は，グラフィックスシステムがその文字をどう描けばよいかを示す一連のベクトルによって定義されています。

ストロークフォントを使う利点は，大きな文字を描こうとする際にはっきりします。ストロークフォントはベクトルで定義されているので，フォントを拡大しても，高い解像度と品位を保つことができます。一方ビットマップフォントを拡大すると，フォントの行列がスケーリング因子倍されます。すなわち，スケーリング因子が大きくなると，文字の解像度は粗くなります。小さな文字には，ビットマップフォントで充分ですが，大きなテキストの場合にはストロークフォントの方を選ぶべきです。なお，漢字が使用できるのはビットマップフォントだけです。

グラフィックステキストを出力するには，**outtext** あるいは **outtextxy** を呼び出します。出力テキストの位置合わせ(テキストのどこをCPに合わせて出力するか)を制御するには **settextjustify** を用います。文字のフォント，向き（水平，垂直)，サイズ（スケール）は **settextstyle** を使って選択します。テキストに関する現在の設定を知るには **gettextsettings** を呼び出しますが，これはフォント，位置合わせ，サイズ，向きに関する情報を **textsettingstype** 構造体に返します。**setusercharsize** を使うと，ストロークフォントの文字幅と高さを変更できます。

クリッピングがオンになっている場合，**outtext** あるいは **outtextxy** によってテキスト文字列が出力されるとき，出力はビューポート境界でクリッピングされます（ビューポート境界の外にはみだす部分はカットされます)。このとき，ビットマップフォントは文字単位でクリッピングされます（文字の一部がビューポート境界の外に出てしまう場合，その文字はまったく出力されません)。またストロークフォントはピクセル単位でクリッピングされます（文字の一部がビューポート境界の外に出る場合，その文字は境界の内側の部分だけが表示されます)。クリッピングがオフの場合には，（ビューポートとは関係なく）出力は画面境界で同じようにクリッピングされます（ビットマップフォントは文字単位，ストロークフォントはピクセル単位)。

現在の設定における特定のテキスト文字列の画面上での大きさを知るには，**textheight**（文字列の高さをピクセル単位で測る）や **textwidth**（幅をピクセル単位で測る）を呼び出します。これによって，出力しようとする文字列の画面上での大きさを，実際に出力を行なう前に知ることができます。

デフォルトの8×16 (8×8) ビットマップフォントはグラフィックスシステムにあらかじめ組み込まれているので，実行時にはいつでも使うことができます。ストロークフォント

は個別に.CHRファイルに保存されているので，実行時にロードするか，あるいは（BGIOBJユーティリティで）.OBJファイルに変換して.EXEファイルにリンクするようにします。

通常**settextstyle**ルーチンは，フォント用にメモリを割り当てて適切な.CHRファイルをディスクから読み込むことによって，フォントのロードを行ないます。このように動的にフォントをロードする方法の他に，フォントファイルを（場合によっては複数個）オブジェクトファイルに変換して，実行可能なプログラムファイルに直接リンクする方法があります。これを行なうには，まず（BGIOBJユーティリティを使って）.CHRファイルを.OBJファイルに変換し，次に，プログラムのソースコード中（**settextstyle**の呼び出しより前）に**registerbgifont**の呼び出しを置いて，文字フォントを登録します。そして，プログラムを作成するときに，登録するストロークフォントに対応する.OBJファイルをリンクします。

注意：**registerbgifont**の使用には，やや高度なプログラミングテクニックを要するので，経験の浅いプログラマには薦められません。この関数は，付録Dの「**BGIOBJ**」の節でもう少し詳しく説明を行なっています。

カラーの制御

ここでは，カラー制御関数を簡単にまとめておきます。

■カラー情報を得る：

getbkcolor	カレントバックグラウンドカラーを返す
getcolor	カレントドロウカラーを返す
getdefaultpalette	パレット定義の構造体を返す
getmaxcolor	カレントグラフィックスモードで使用可能なカラーの最大値を返す
getpalette	カレントパレットとそのサイズを返す
getpalettesize	パレット参照テーブルのサイズを返す

■カラーの設定：

setallpalette	すべてのパレットカラーの変更
setbkcolor	カレントバックグラウンドカラーの設定
setcolor	カレント描画カラーの設定

setpalette	パレットカラーを指定したカラーに変更する
setrgbpalette	ハードウェアパレットのRGBの各々の強度を設定する

これらのカラー制御関数がどのように動くかについてまとめる前に，まず画面上の色が実際にどのように作られるかについて，基本的な説明を行ないます。

ピクセルとパレット

グラフィックス画面はピクセルの並びで構成されており，各ピクセルは画面上の1つの（色つきの）点を構成します。ピクセルの値は実際の色を直接指定しているのではなく，**パレット**と呼ばれる色のテーブルへのインデックスになっています。与えられたピクセルの値に対応するパレット項目の中に，そのピクセルの色についての情報が含まれています。

このような間接的なやり方には，多くの意味が含まれています。ハードウェアがたくさんの色を表示する能力をもっていたとしても，一度に表示できるのはたいていそれらの中の一部だけです。一度に表示できる色の数はパレットの項目数（**パレットサイズ**）と同じです。たとえば，PC-9801の4096色ボードを使えば，ハードウェアとしては4096色の表示が可能ですが，一度に表示できるのはその中の16色だけです。すなわちパレットサイズは16になります。

パレットのサイズは，ピクセルの値がとり得る範囲，つまり0～(パレットの大きさ-1)を規定することになります。**getmaxcolor** 関数は，現在のグラフィックスドライバとモードにおいて可能な最大のピクセル値（パレットの大きさ-1）を返します。

Turbo Cのグラフィックス関数の説明では，カレントドロウカラー，フィルカラー，ピクセルカラーなどのように「**カラー**」という用語を多用しています。ここでいうカラーというのは，実際にはピクセルの値，つまりパレットへのインデックスです。画面上の実際の色を決定するのはパレットだけです。パレットを操作することによって，画面上に実際に表示される色を，ピクセルの値（ドロウカラー，フィルカラーなど）を変更せずに変えることができます。

バックグラウンドカラーとドロウカラー

バックグラウンドカラーはピクセル値0に対応します。ある領域がバックグラウンドカラーでクリアされるときには，その領域のピクセルが0に設定されるだけです。

ドロウカラーとは，線が描かれるときにピクセルにセットされる値のことです。ドロウカラーは **setcolor**(*n*)で選択されます。*n* は現在のパレットに有効なピクセル値です。

PC-9801上でのカラー制御

グラフィックハードウェアの違いにより，4096色ボードがあるときとないときでは実際のカラー制御の方法が異なります。そこで，ここではそれらを別々に説明します。

PC-9801 8色モード

4096色ボードがない（最初期型のPC-9801やPC-9801E/F/Mなど）場合は，定義済みの8色のパレットから1つを選ぶことができます。

カレントパレットの設定は**getpalette**で調べることができます。この関数は構造体にパレットの大きさ（16）と実際のパレット中の項目（パレットに納められているハードウェアカラー番号）を書き込む働きを持っています。パレットの項目は，**setpalette**によって一つ一つ変更することができます。また**setallpalette**によって全部を同時に変更することもできます。PC-9801の8色モードで使用できるカラーを次の表に示します。これはGRAPHICS.Hで定義されているものです。

数値	シンボリック名	実際の表示色
0	BLACK	黒
1	BLUE	青
2	GREEN	緑
3	CYAN	水色
4	RED	赤
5	MAGENTA	紫
6	BROWN	黄
7	LIGHTGRAY	白
8	DARKGRAY	黒
9	LIGHTBLUE	青
10	LIGHTGREEN	緑
11	LIGHTCYAN	水色
12	LIGHTRED	赤
13	LIGHTMAGENTA	紫
14	YELLOW	黄

15	WHITE	白

PC-9801 16色モード

このモードでは，パレットには16項目が含まれており，それぞれ下の表のように実際の色がセットされています。

カレントパレットの設定は **getpalette** で調べることができます。この関数は構造体にパレットの大きさ（16）と実際のパレット中の項目をセットします。実際のパレットのカラーは **setpalette** により一つ一つ変更することができます。また **setallpalette** によって全部を同時に変更することもできます。

数値	シンボリック名	実際の表示色
0	BLACK	黒
1	BLUE	青
2	GREEN	緑
3	CYAN	水色
4	RED	赤
5	MAGENTA	紫
6	BROWN	茶
7	LIGHTGRAY	明るい灰色
8	DARKGRAY	濃い灰色
9	LIGHTBLUE	明るい青
10	LIGHTGREEN	明るい緑
11	LIGHTCYAN	明るい水色
12	LIGHTRED	明るい赤
13	LIGHTMAGENTA	明るい紫
14	YELLOW	黄
15	WHITE	白

このように8色モードのときとテーブルは同じです。ただし8色モードの場合は8以上を指

定してもすべてBGIが8色におさまるように変換してしまいます。

さらにこのモードでは**setrgbpalette**を使うことで，4096色から任意の16色をパレットにセットすることができます。

IBM PC CGAでのカラー制御

グラフィックスハードウェアの違いから，実際にどのようにカラーを制御するかは，CGAとEGAでは異なってくるため，ここでは別々に説明しています。AT&Tドライバ，およびMCGAの解像度の低いドライバでのカラー制御は，CGAのカラー制御と似たものになります。

CGAでは，4色が使用できる低解像度と2色が使用できる高解像度のどちらでグラフィックスを表示するかを選択することができます。

CGA低解像度モード

低解像度モードでは，定義済みの4つある4色のカラーパレットの中からカラーを選択します。どのパレットに対しても，ユーザが設定できるのは最初の項目だけで，項目1,2,3は固定です。最初のパレット項目（カラー0）はバックグラウンドカラーです。このバックグラウンドカラーは，16個の使用可能なカラーの中から任意にセットすることができます(次のページの表を参照してください)。

どのパレットを使うかは，モード(CGAC0，CGAC1，CGAC2，CGAC3)を選択することによって決まります。各モードは，それぞれカラーパレット0からカラーパレット3を使用します。各パレットは次の表のようになっています。CGAのドロウカラーおよび同じ値の定数はgraphics.hで定義されています。

	カラー番号（ピクセル値）に割り当てられている定数		
パレット番号	1	2	3
0	CGA_LIGHTGREEN	CGA_LIGHTRED	CGA_YELLOW
1	CGA_LIGHTCYAN	CGA_LIGHTMAGENTA	CGA_WHITE
2	CGA_GREEN	CGA_RED	CGA_BROWN
3	CGA_CYAN	CGA_MAGENTA	CGA_LIGHTGRAY

上のカラーのうちの1つをCGAドロウカラーとして割り当てるためには，そのカラー番号または対応する定数名を引数として**setcolor**を呼び出します。たとえば，パレット3を使用中で，ドロウカラーとして水色（シアン）を使いたい場合には次のようにします。

```
setcolor(1);     または     setcolor(CGA_CYAN);
```

CGAのバックグラウンドカラーとして使用できるカラーはgraphics.hで定義されており，次の表のようになっています。

数値	シンボリック定数
0	BLACK
1	BLUE
2	GREEN
3	CYAN
4	RED
5	MAGENTA
6	BROWN
7	LIGHTGRAY
8	DARKGRAY
9	LIGHTBLUE
10	LIGHTGREEN
11	LIGHTCYAN
12	LIGHTRED
13	LIGHTMAGENTA
14	YELLOW
15	WHITE

この中からCGAのバックグラウンドカラーを割り当てるには，**setbkcolor**(*color*)を使います。*color*には上の表の数値または定数名を指定します。CGAにおいては，このカラーはピクセル値（パレットインデックス）ではなく，パレットの最初の項目にセットされる実際の色を直接示していることに注意してください。

CGA 高解像度モード

高解像度モード（640×200）では，CGA は2色を表示します。バックグラウンドの黒と，フォアグラウンドのカラーです。ピクセルがとり得る値は0か1のどちらかです。CGA そのものの特殊性のために，フォアグラウンドカラーは，実際にはハードウェアがバックグラウンドカラーであると考えているものです。したがって，フォアグラウンドカラーをセットするには **setbkcolor** ルーチンを使用します（奇妙ですが事実です）。

フォアグラウンドに指定できるカラーは，前の（バックグラウンドカラーの）表の通りです。CGA はこのカラーを使って，値1を持つすべてのピクセルを表示します。

このような動作をするモードは，CGAHI，MCGAMED，MCGAHI，ATT400MED，および ATT400HI です。

CGA パレットルーチン

CGA のパレットはあらかじめ決まっているので，CGA 上では **setallpalette** ルーチンを使ってはなりません。また，**setpalette**(*index*, *actual _color*) も，*index* ＝ 0の場合以外は使用してはいけません（*index* ＝ 0で **setpalette** を呼び出すことは，CGA のバックグラウンドカラーを *actual _color* にセットする方法の代替になります)。

EGA および VGA でのカラー制御

EGA では，パレットの項目は16あり，すべての項目はトータル64色の中からセットすることができます。

現在のパレットは，**getpalette** によって取り出すことができます。**getpalette** は，パレットサイズ（16）とパレットの実際の項目（パレットに格納されているハードウェアカラー番号）の配列を構造体に入れて返します。**setpalette** を使うとパレット項目を個々にセットすることができ，**setallpalette** を使えば全項目を一度にセットすることもできます。

デフォルトの EGA パレットは，前の表に示した CGA の16色に対応しています。黒が項目0，青が項目1，・・・・・，白が項目16です。ハードウェアカラー値に対応する定数は，graphics.h の中で定義されており，EGA_BLACK，EGA_WHITE などの名前になっています。これらの値は **getpalette** で取り出すことができます。

EGA 上では，**setbkcolor**(*color*) ルーチンは，CGA 上とは異なる動作を示します。EGA では，**setbkcolor** は *color* が持っている実際のカラー値をパレットの項目0にコピーします。

カラーに関する限り，VGA ドライバは EGA ドライバと同じように動作します。もちろん VGA はより高い解像度（およびより小さなピクセル）を持っています。

グラフィックスモードにおけるエラー処理

ここではグラフィックスモードにおけるエラー処理関数について簡単に説明を行ないます。

grapherrormsg	指定されたエラーコードのエラーメッセージ文字列を返す
graphresult	エラーが生じた最後のグラフィック操作のエラーコードを返す

グラフィックスライブラリ関数が呼び出されたときにエラーが発生した場合（たとえば，**settextstyle** で指定されたフォントが見つからなかった場合），内部エラーコードがセットされます。エラーが発生した最後のグラフィックス操作に対するそのエラーコードは，**graphresult** を呼び出すことによって調べることができます。次のようなエラーリターンコードが定義されています。

エラーコード	定数	対応するエラーメッセージ文字列/意味
0	grOK	No error エラーなし
-1	grNoInitGraph	(BGI) graphics not installed (use initgraph) グラフィックス(BGI)がインストールされていない（initgraph を使用せよ）
-2	grNotDetected	Graphics hardware not detected グラフィックハードウェアが検出できない
-3	grFileNotFound	Device driver file nor found デバイスドライバファイルが見つからない
-4	grInvalidDriver	Invalid device driver file 正しくないデバイスドライバファイル

-5	grNoLoadMem	Not enough memory to load driver ドライバをロードするのにメモリが不足した
-6	grNoScanMem	Out of memory in scan fill スキャンフィルでメモリが不足した
-7	grNoFloodMem	Out of memory in flood fill フラッドフィルでメモリが不足した
-8	grFontNotFound	Font file not found フォントファイルがみつからない
-9	grNoFontMem	Not enough memory to load font フォントをロードするのにメモリが不足した
-10	grInvalidMode	Invalid graphics mode for selected driver 選んだドライバに対してグラフィックスモードが正しくない
-11	grError	Graphics error グラフィックスエラー
-12	grIOerror	Graphics I/O error グラフィックスI/Oエラー
-13	grInvalidFont	Invalid font file フォントファイルが正しくない
-14	grInvalidFontNum	Invalid font number フォント番号が正しくない
-15	grInvalidDeviceNum	Invalid device number デバイス番号が正しくない
-18	grInvalidVersion	Invalid version of file ファイルのバージョンが正しくない

grapherrormsg(graphresult())を呼び出すと，上の表のエラー文字列が返されます。

エラーリターンコードは保持され，グラフィックス関数がエラーを起こしたときだけ変更されます。エラーリターンコードは，**intgraph**の実行が成功した場合と**graphresult**が呼び出された場合にのみ0にリセットされます。このため，どのグラフィック関数がどのエラーを起こしたかを知りたい場合は，**graphresult**の値をテンポラリ(一時的な)変数に代入して，それを調べる必要があります。

状態の問い合わせ

以下はグラフィックスモードの状態問い合わせ関数の要約です。

getarccoords	**arc** または **ellipse** の最後の呼び出し時の座標の情報を返す
getaspectratio	グラフィックス画面のアスペクト比を返す
getbkcolor	カレントバックグラウンドカラーを返す
getcolor	カレントドロウカラーを返す
getdrivername	カレントのグラフィックスドライバ名を返す
getfillpattern	ユーザ定義のフィルパターンを返す
getfillsettings	カレントフィルパターンとフィルカラーについての情報を返す
getgraphmode	カレントグラフィックスモードを返す
getlinesettings	カレントラインスタイル，パターン，幅を返す
getmaxcolor	現在の有効なピクセル値の最大値を返す
getmaxmode	与えられたドライバに有効なモードの最大値を返す
getmaxx	現在の *x* の分解能を返す
getmaxy	現在の *y* の分解能を返す
getmodename	カレントのグラフィックスモード名を返す
getmoderange	グラフィックスドライバのモード範囲を返す
getpalette	カレントパレットとその大きさを返す
getpixel	*x*，*y* のピクセルの色を返す
gettextsettings	カレントテキストフォント，向き，大きさ，桁揃えを返す
getviewsettings	カレントビューポートについての情報を返す
getx	現在位置（CP）の *x* 座標を返す
gety	現在位置（CP）の *y* 座標を返す

Turbo C のグラフィックス関数の7つのカテゴリには，少なくとも1つの状態問い合わせ関数があります。この関数についてはそれぞれのカテゴリのところで述べていますが，ここでもそれらについて説明しておきます。Turbo C グラフィックスの状態問い合わせ関数には“**get**”で始まる名前がついています（エラー処理関数のカテゴリを除きます)。これら

の関数のうち，いくつかのものは引数をとらず，要求された情報を表わす1個の値を返します。他の関数は，graphics.h で定義されている構造体へのポインタを受け取り，その構造体に必要となる情報を書き込み，関数値としては何も返さないものです。

グラフィックスシステム制御カテゴリの状態問い合わせ関数としては，**getgraphmode**, **getmaxmode**, **getmoderange** があります。**getgraphmode** はカレントグラフィックスドライバとモードを表わす1個の整数を返します。**getmaxmode** は与えられたグラフィックスドライバにおける最大のモード値を返します。**getmoderange** は与えられたグラフィックスドライバがサポートしているモードの範囲を返します。**getmaxx** と **getmaxy** はそれぞれカレントグラフィックスモードの X-Y 画面座標の最大値を返します。

ドロウおよびフィルに関する状態問い合わせ関数には，**getarccoords**, **getaspectratio**, **getfillpattern**, **getfillsettings**, **getlinesettings** があります。**getarccoords** は構造体に **arc** または **ellipse** の最後の呼び出し時の座標を書き込みます。**getaspectratio** はカレントモードのアスペクト比を与えます。これは，グラフィックスシステムが円を本当に丸く描くために用いられるものです。**getfillpattern** はカレントのユーザ定義のフィルパターンを返します。**getfillsettings** は構造体にカレントフィルパターンとフィルカラーを書き込みます。**getlinesettings** は構造体にカレントラインスタイル（実線，破線など），ライン幅（細線または太線），ラインパターンを書き込みます。

画面およびビューポート操作カテゴリの状態問い合わせ関数には，**getviewsettings**, **getx**, **gety**, **getpixel** があります。ビューポートを定義したとき，**getviewsettings** を呼び出すことによって，その画面上の絶対座標およびクリッピングがアクティブかどうかを知ることができます。この関数は構造体にこれらの情報を書き込みます。**getx** と **gety** は，それぞれ CP の(ビューポート相対の) *x* 座標と *y* 座標を返します。**getpixel** は指定されたピクセルのカラーを返します。

グラフィックスモードのテキスト出力関数カテゴリには，包括的な状態問い合わせ関数が1つ含まれています。**gettextsettings** がそれで，構造体にカレント文字フォント，テキスト表示の向き（水平または垂直），文字拡大因子，テキスト文字列の位置合わせ（水平方向と垂直方向の両方）についての情報を書き込みます。

カラー制御関数カテゴリには，4つの状態問い合わせ関数があります。**getbkcolor** はカレントバックグラウンドカラーを返します。getcolor はカレントドロウカラーを返します。**getpallete** は構造体にカレントドロウパレットの大きさとそのパレットの内容を書き込みます。**getmaxcolor** はカレントグラフィックスドライバとモードにおいて有効なピクセル値の最大値（パレットの大きさ-1）を返します。

getmodename と **getdrivername** はそれぞれ，与えられたモード（番号）の名前，およびカレントグラフィックスドライバの名前を返します。

第9章

Turbo Pascalプログラマのために

この章に入っていく前に，ぜひ第6章と第7章を，ざっとでもいいので見直してください。Cがプログラミングの基本要素をどのように実現しているかを理解してほしいのです。この章でも同じような点について述べていますが，ここでは触れられない詳細な点が前の2章では述べられているのです。

おおまかに言うと，Pascalはかなり厳密で構造化された言語であるのに対し，Cはかなり自由でフレキシブルな言語です。しかし，Cは注意して使うべきものであり，経験を積んだプログラマには大きな力を与える自由度が，初心者にとっては大きなトラブルの元となってしまうことがあります。PascalはCにくらべると面倒見がよく，プログラミングの基礎を学習するには向いていると言えます。

Turbo CとTurbo Pascalはともに，このC-Pascal言語スペクトルの中心に向かって進んでいます。Turbo CはCに構造を与え，Turbo PascalはPascalに柔軟性を与えています。

この章では……

この章では，Cとその多くの機能についての包括的な説明は行ないません。この章の目的はTurbo Pascalのプログラマが，Turbo Cを使ってCのプログラムを書くのを助けることにあります。Cに関する専門的知識や洞察力は，時間をかけてプログラミングを実践し，数百行にわたるようなソースコードを書くことによって得られるはずです。

ここでは，Turbo PascalとTurbo Cのプログラミングにおいて，類似している点と異なっている点を明確にしていきます。最初は，プログラムの構造やプログラミングの要素など基本的なことから始めます。そのあと，典型的な例を使ってデータ構造について説明します。最後に，プログラミングの上で注意すべき問題点について説明し，またCを学ぼうとしているPascalのプログラマがおちいりやすい落し穴を紹介しています。

この章を通して，プログラム例を使って指摘したい点を明らかにしていきます。各サン

プルは，左側に Turbo Pascal，右側に同じ機能の Turbo C プログラムを並べてあります。

プログラムの構造

ご存知のように，Turbo Pascal におけるプログラムの構造は次のような形式をとっています。

```
program プログラム名;
  const              {                      }
  type               {  宣言部              }
  var                {  (自由な構成でよい)  }
  手続き-関数        {                      }

begin { プログラムの本体    }
  文;
end.  { プログラムの終わり }
```

プログラムのメインボディが実行されます。メインボディがさらに手続きや関数を呼び出していれば，それも実行されます。識別名（定数，型，変数，手続き，関数）はすべて使用する前に宣言しておかなければなりません。手続きと関数はほぼ同じ形式で構成されています。

Cにおけるプログラムの構造はもう少しフレシキブルです。

```
プリプロセッサコマンド    /*                  */
型の定義                  /*                  */
関数プロトタイプ          /*  自由な構成でよい  */
変数                      /*                  */
関数                      /*                  */
```

この中で関数は次のような構造をとっています。

```
型  関数名 (引数の宣言)
{
      ローカルな宣言
      文
}
```

宣言する関数の中には，1つだけ **main** という名前の関数が存在しなければなりません。これがプログラムの“メインボディ”です。いいかえると，Turbo C のプログラムの実行が始まると，まず **main** が呼び出され，それがさらに他の関数を呼び出すというわけです。Cのプログラムはいくつかの関数が集まってできています。関数の中には void 型，つまり何も値を返さない関数もあり，これは Pascal でいえば手続きに当たります。また Pascal とは異なり，関数が返す値を無視することもできます。

プログラム例

次に2つのプログラムを示します。1つはTurbo Pascal，もう1つはTurbo Cで書いたものです。両者の間には，似ている点も異なっている点もあります。

Turbo Pascal

```
program MyProg;
var
  I,J,K Integer;

function Max(A,B:Integer) : Integer;
begin
  if A > B
    then Max := A
    else Max := B

end;
{ 関数 Max 終わり }

procedure Swap(var A,B : Integer);
var
  Temp : Integer;
begin
  Temp := A; A := B; B := Temp
end;
{ 手続き Swap 終わり }


begin    { MyProg のメインボディ }
  I := 10; J := 15;
  K := Max(I,J);
  Swap(I,K);
  Write('I = ',I:2,' J = ',J:2);
  Writeln(' K = ',K:2)
end.
{ プログラム MyProg 終わり }
```

Turbo C

```
int  i,j,k;

int  max(int a, int b)
{
   if (a>b)
      return(a);
   else
      return(b);
}
/* max() 終わり */

void swap(int *a, int *b)
{
   int    temp;

   temp = *a; *a = *b; *b =temp;
}
/* swap() 終わり */

main()
{
   i = 10; j = 15;
   k = max(i,j);
   swap(&i,&k);
   printf("i = %2d j = %2d",i,j);
   printf(" k = %2d¥n",k);
}
/* main 終わり */
```

*i,j,k*は，グローバル変数とするよりは，**main**の中で宣言した方がよいでしょう。そうすると，グローバル変数を関数の中から直接変更してしまう可能性を消すことができるからです。またグローバル変数は，プログラムの実行中はずっと存在してしまいます。

右側のＣのプログラムはやや奇妙に見えるかもしれませんが，この章を読み終える頃には，すっかり落着いて見ることができるでしょう。実際もっと奇妙なものを書くこともできます。

プログラミングの要素の比較

第6章で，プログラミングの7つの基本要素について述べました。つまり，出力，データ型，演算，入力，条件による実行，ループ，サブルーチンの7つです。これをもう一度ながめて，Pascal と C は，どこが似ているのか，またどこが異なっているのかを調べていくことにします。

出力

Turbo Pascal の出力命令の中心は ***Write*** と ***Writeln*** です。Turbo C では，たくさんの命令が存在し，好きなものを選ぶことになります。一番よく使われるのは（オーバーヘッドも一番大きいのですが）**printf** で，次のような書式をとります。

```
printf(書式文字列, 項, 項, ...);
```

書式文字列は，**文字列リテラル**または**文字列変数**のいずれかです（文字列は二重引用符で囲むことを憶えておいてください）。**項**はオプションで，変数や式などを指定し，書式文字列の中の書式コマンドに対応するものです。詳しくは第6章を見てください。C では，改行を行なう（= ***Writeln***）には，エスケープシーケンス¥n（改行）を書式文字列の最後に置きます。

次に Turbo Pascal の文の例と，それに対応する（あるいは近い）C の文をいくつか示します。

Turbo Pascal	Turbo C
`var`	
`  A,B,C : Integer;`	`int    a,b,c;`
`  Amt   : Real;`	`float  amt;`
`  Name  : String[20];`	`char   name[21]; /* または *name */`
`  Ans   : Char;`	`char   ans;`
`Writeln('Hello, world.');`	`printf("Hello, world.¥n");`
`Write('What''s your name? ');`	`printf("What's your name? ");`
`writeln('"Hello," said John');`	`printf("¥"Hello,¥" said John¥n");`
`Writeln(A,' + ',B,' = ',C);`	`printf("%d + %d = %d¥n",a,b,c);`
`Writeln('You owe us $',Amt:6:2);`	`printf("You owe us $%6.2f¥n",amt);`
`Writeln('Your name is ',Name,'?');`	`printf("Your name is %s?¥n",name);`
`Writeln('The answer is ',Ans);`	`printf("The answer is %c¥n",ans);`
`Write(' A = ',A:4);`	`printf(" a = %4d",a);`
`Writeln(' A*A = ',(A*A):6);`	`printf(" a*a = %6d¥n",a*a);`

この他に知っておくとよいCの出力ルーチンは**puts**と**putchar**です。**puts**は，1個の文字列を引数にとり，その文字列を書き出して最後に改行を付け加えます。**putchar**はもっと簡単で，1個の文字を書くだけです。以下の命令はそれぞれ同じ働きをします。

`Writeln(Name);`	`puts(Name);`
`Writeln('Hi, there!'); Writeln;`	`puts("Hi, there!¥n");`
`Write(Ch);`	`putchar(ch);`

データ型

Turbo Pascalのデータ型のほとんどは，対応するものがTurbo Cに存在します。Cはたくさんのデータ型を持っており，整数型や浮動小数点型には大きさ（バイト数）が異なる数種のものが用意されています。さらに修飾子signedやunsignedもあります。次にPascalとCのデータ型のおおまかな対応表を示します。

Turbo Pascal			Turbo C		
***shortint**	(1バイト)	-128～127	char	(1バイト)	-128～127
char	(1バイト)	chr(0～255)			
byte	(1バイト)	0～255	unsigned char	(1バイト)	0～255
integer	(2バイト)	-32768～32767	short	(2バイト)	-32768～32767
			int	(2バイト)	-32768～32767
***word**	(2バイト)	0～65536	unsigned int	(2バイト)	0～65535
***longint**	(4バイト)	-2^{31}～2^{31}-1	long	(4バイト)	-2^{31}～2^{31}-1
			unsigned long	(4バイト)	0～(2^{32}-1)
***single**	(4バイト)	±3.4E±308	float	(4バイト)	±3.4E±308
real	(6バイト)	1E-38～1E+38			
***double**	(8バイト)	±1.7E±308	double	(8バイト)	±1.7E±308
***extended**	(10バイト)	±3.4E-4932 ～1.1E+4932	long double	(10バイト)	±3.4E-4932 ～1.1E+4932
boolean	(1バイト)	False, True	0 ＝ false, 0以外 ＝ true		

* は Turbo Pascal 4.0で新たに追加されたデータ型です。

booleanに対応するデータ型がCには存在しないことに注意してください。論理型の値をとる式は，0のときに**偽**，0でないときに**真**と解釈されるようになってます。

以上のデータ型に加えて，Turbo Cは**列挙型**のデータ型をサポートします。しかしPascalとは違って，これは結局のところ，あらかじめ値が与えられている整定数の組を表わすものであり，すべての整数型と完全にコンパチブルとなります。

Turbo Pascal	Turbo C
<pre>type Days = (Sun, Mon, Tues, Wed, Thurs, fri, Sat); var Today : Days;</pre>	<pre>enmu days { Sun, Mon, Tues, Wed, Thurs, Fri, Sat }; enum days today;</pre>

演算

Turbo C は Turbo Pascal の演算子すべてに対応するものを持ち，その他にさらにいくつかの演算子を持っています。

2つの言語の間で大きく異なる演算の1つに代入があります。Pascal では代入（:=）は文であるのに対し，C では代入（=）は式の中で使うことができる演算子です。

表 6.1 に，Turbo Pascal と Turbo C における演算子の比較を示します。演算子は優先順に並べてあり，グループ内では優先順位は同じです。

表 9.1　Pascal と C の演算子

<table>
<tr><th>演算子</th><th>Pascal</th><th>C</th></tr>
<tr><td>単項マイナス</td><td>A := -B;</td><td>a = -b;</td></tr>
<tr><td>単項プラス</td><td>A := +B;</td><td>a = +b;</td></tr>
<tr><td>論理否定</td><td>not flag</td><td>!flag</td></tr>
<tr><td>ビットごとの補数</td><td>A := not B;</td><td>a = ~b;</td></tr>
<tr><td>アドレス</td><td>A := Addr(B);</td><td>a = &b;</td></tr>
<tr><td>ポインタ参照</td><td>A := IntPtr^;</td><td>a = *intptr;</td></tr>
<tr><td>大きさ</td><td>A := SizeOf(B);</td><td>a = sizeof(b);</td></tr>
<tr><td>インクリメント</td><td>A := Succ(A);</td><td>a++　　++a</td></tr>
<tr><td>デクリメント</td><td>A := Pred(A);</td><td>a--　　--a</td></tr>
<tr><td>乗算</td><td>A := B * C;</td><td>a = b * c;</td></tr>
<tr><td>整数除算</td><td>A := B div C;</td><td>a = b / c</td></tr>
<tr><td>浮動小数点除算</td><td>X := B / C;</td><td>b = d / c;</td></tr>
<tr><td>剰余</td><td>A := B mod C;</td><td>a = b % c;</td></tr>
<tr><td>加算</td><td>A := B + C;</td><td>a = b + c;</td></tr>
<tr><td>減算</td><td>A := B - C;</td><td>a = b - c;</td></tr>
<tr><td>右にシフト</td><td>A := B shr C;</td><td>a = b >> c;</td></tr>
<tr><td>左にシフト</td><td>A := B shl C;</td><td>a = b << c;</td></tr>
<tr><td>より大きい</td><td>A > B</td><td>a > b</td></tr>
<tr><td>より大きいまたは等しい</td><td>A >= B</td><td>a >= b</td></tr>
<tr><td>より小さい</td><td>A < B</td><td>a < b</td></tr>
<tr><td>より小さいまたは等しい</td><td>A <= B</td><td>a <= b</td></tr>
<tr><td>等しい</td><td>A = B</td><td>a == b</td></tr>
<tr><td>等しくない</td><td>A <> B</td><td>a != b</td></tr>
<tr><td>ビットごとのAND</td><td>A := B and C;</td><td>a = b & c;</td></tr>
<tr><td>ビットごとのXOR</td><td>A := B xor C;</td><td>a = b ^ c;</td></tr>
<tr><td>ビットごとのOR</td><td>A := B or C;</td><td>a = b | c;</td></tr>
<tr><td>論理的AND</td><td>Flag1 and Flag2</td><td>flag1 && flag2</td></tr>
<tr><td>論理的OR</td><td>Flag1 or Flag2</td><td>flag1 || flag2</td></tr>
<tr><td>代入</td><td>A := B;
A := A <op> B;</td><td>a = b;
a <op>= b;</td></tr>
</table>

Pascal と C の演算子では，重要な差異がいくつかあります。まずインクリメント(++)とデクリメント（−−）演算子は，変数名の前に置く場合と，後ろに置く場合があります。変数の前に置くと，式が評価される前にインクリメント（またはデクリメント）が行なわれます。後ろに置くと，まず式が評価され，その後インクリメント（デクリメント）されます。

次に，Cの論理演算子は**ショートサーキット**（短絡回路）をとることがあります。つまり，最初の項によって式全体の真偽が決定できる場合は，2番目以降の項の評価は行なわないのです。したがって，Cでは次のように書くことができます（*limit* は配列 *list* の添字の最大値です）。

```
while (i <= limit && list[1] != 0) ... ;
```

最初の項（i <= limit）が偽であれば，式全体の値は偽に決まっているので，2番目の項（list[i] != 0）は評価されません（この式が評価されると添字が宣言された範囲を越えてしまうのですが）。

第3に，Cでは次のような一般的な式は，

```
A = A <演算子> B
```

次のように書くこともできます（演算子は，&&と｜｜を除く二項演算子です）。

```
A <演算子>= B
```

したがって，たとえば次の2つの式は同じ意味を持つことになります。

```
A = A * b
A *= B
```

入力

Turbo Pascal では基本的な入力命令は ***Read***() で，そのバリエーションがいくつかあります（***Readln***()，***Read***(***f,***)，など）。Turbo C ではキーボードから入力を行なう関数の中心は **scanf** で，書式は次のようになっています。

```
scanf(書式文字列, アドレス1, アドレス2, ...);
```

書式文字列は **printf** と同様で，書式指定子を含む文字列です。またアドレス1以降は，入力されたデータを格納するアドレスです。したがって，アドレス of 演算子（&）がよく使われることになります。他にもよく使う関数がいくつかあります。**gets** は**リターン**が押されるまでの文字列全体を読み込み，**getch** はキーボードからエコーなしで1文字だけを読み込みます。

Pascal と C の，対応する入力命令を示しましょう。

Turbo Pascal	Turbo C
`Readln(A,B);`	`scanf("%d %d",&a,&d);`
`Readln(Name);`	`scanf("%s",name);` `/* または gets(name); */`
`Readln(X,A);`	`scanf("%f %d",&x,&a);`
`Readln(Ch);`	`scanf("%c",ch);`
`Ch := ReadKey;` `{ TP 3.0 では Read(Kbd,Ch); }`	`ch = getch();`

文字列を読み込む際の，**scanf** と **gets** の働きの違いに注意してください。まず **scanf** は，ホワイトスペース（空白，タブ，改行）が入力されるまでのすべての文字を読み込みます。一方 **gets** は，**リターン**を押すまで入力された文字をすべて（空白やタブも）読み込みます。

ブロック文

Pascal も C も，**ブロック文**という概念をもっています（文がいくつか集まったもので，単一の文が置ける場所であればどこにでも置くことができます)。Pascal では，ブロック文は次のような形式になっています。

```
begin 文; 文; ... ; 文 end;
```

C では，次のようになります。

```
{ 文; 文; ... 文; }
```

かなり似ていますが，違っている点が2つあります。

■ Pascal ではブロックの最後の文の後ろにセミコロンを置く必要はありませんが，C では必要です。

■ C では，ブロックを囲む右カッコ } の後ろにセミコロンは置きませんが，Pascal では（**end** の後に）置かなければならないことがあります。

条件による実行

Pascal も C も**条件による実行構文**を2つ持っています。**if/then/else** 文と **case** 文です。**if/then/else** は，両者でよく似ています。

Turbo Pascal	Turbo C
<pre>if 論理式 then 文 else 文</pre>	<pre>if (式) 文; else 文;</pre>

Pascal と C どちらにおいても，**else** 節はオプションです。文には上で述べたブロック文も置くことができます。しかし，ここで重要な違いがいくつかあります。

- ■C では，式は論理型でなくても，0か0以外の値かがはっきりすればよい。0は偽，0以外は真とみなされる。
- ■C では，式はカッコで囲まなければならない。
- ■C では，予約語 **then** はない。
- ■C では，文の後ろには必らずセミコロンが必要である（もちろんブロック文の最後にはつけない）。

Pascal と C の **if/then/else** の例を示します。

Turbo Pascal	Turbo C
<pre>if B = 0 then Writeln('C is undefined') else begin C := A div B; Writeln('C = ',C) end;</pre>	<pre>if (b == 0) puts("c is undefined"); else { c = a / b; printf("c = %d¥n",c); }</pre>
<pre>C := A * B; if C <> 0 then C := C + B else C := A;</pre>	<pre>if ((c = a * b) != 0) c += b; else c = a;</pre>

case 文も，Pascal と C どちらでも使用することができます (C では switch 文と呼ばれます)。両者には重要な違いがいくつかあります。

次に一般的形式を示します。

Turbo Pascal	Turbo C
<pre>case 式 of リスト : 文; リスト : 文; ... リスト : 文; else 文 end;</pre>	<pre>switch (式) { case 項 : 文; case 項 : 文; ... case 項 : 文; default : 文; }</pre>

表面的な違いの他に，重要な違いもいくつかあります。

まず Pascal では，リストに値の並びを指定することができます。Turbo Pascal では部分範囲 ('A'..'Z' など) も指定できます。C では項には1個の値しか指定できません。どちらの言語でも，順序づけができる定数値しか指定できません。つまり，整数型，文字型，列挙型の値だけです。

次に (これがたいへん重要ですが)，Pascal では，文は1つの文かブロック文で，その実行が終わると **case** 文の残りの部分はスキップされます。C では，文は0個 (つまり何も無い) から複数個の文 (各文はセミコロンで終わる) からなり，その実行が終わっても，制御は

switch 文の終わりに飛ぶのではなく，（途中に case 項があっても）次の文に移ります。break 文にぶつかると，制御は switch 文の終わりに飛びます。case 項は，switch（式）文がどこへ飛ぶかを決めるためのラベルであると考えればよいでしょう。

例を2つ示しましょう。

Turbo Pascal

```
case CH of

  'C' : DoCompile;
  'R' : begin
          if not Compiled
            then DoCompile;
          RunProgram
        end;
  'S' : SaveFile;
  'E' : EditFile;
  'Q' : begin
            if not Saved
            then Savefile
        end;
end;

case Today of
  Mon..Fri : Writeln('go work');
  Sat, Sun : begin
    if Today = Sat then begin
        Write('clean the');
        Write(' yard and ')
      end;

    Writeln('relax')
    end
end;
```

Turbo C

```
switch (ch)
{
   case 'C' : DoCompile(); break;
   case 'R' : if (!compiled)
                   DoCompile();
              RunProgrum();
              break;

   case 'S' : SaveFile(); break;
   case 'E' : EditFile(); break;
   case 'Q' : if (!saved)
                   SaveFile();
              break;

}

switch (today) {
   case Mon :
   case Tue :
   case Wen :
   case Thur:
   case Fri : puts("go work!"); break;
   case Sat : printf("%s", "clean the"
                      " yard and ");
   case Sun : puts("relax!");

}
```

2番目の例を見てください。switch 文の case 項は，実行したいケースのラベルとして働きます。*Mon*～*Thur* の場合は後ろは空になっており，制御は case *Fri*：というラベルのついた文のところまで移動します。そこには break 文があるので，文を実行した後は switch 文の終わりまでスキップすることになります。しかしこのプログラムでは，週末のところで，上で説明した機能をうまく使っています。case *Sat*：では，まず **printf** 文が実行され，次に制御は下に移って **puts** 文が実行されるわけです。

ループ

Pascalと同様，Cも3種類のループを持っています。つまり，while，do..while，forの3つで，それぞれPascalの**while**，**repeat..until**，**for**に対応しています。

while ループ

3つのループの中で，whileループが両言語で一番よく似ているでしょう。書式は次のようになっています。

```
while 論理式 do                while (式)
      文;                            文;
```

どちらの言語も，ループで複数の文を実行したい場合はブロック文を使うことになります。実質的な唯一の違いは，Cの方が，ループを制御する式により柔軟性があることです。次の2つのループを比較してみてください。

```
Ch := ReadKey;
while Ch <> 'q' do begin              while ((ch = getch()) != 'q')
  Write(Ch); Ch := ReadKey                putch(ch);
end;
```

do..while ループ

do..whileループは，Pascalの**repeat..until**に似ています。書式は次のようになります。

```
repeat                              do
   複数個の文                            文;
until 論理式;                        while (式);
```

ただし，両者の間には重要な違いが2つあります。

■ do..whileは，式が真である間（while）ループが実行されるのに対し，**repeat..until**は，論理式が真になるまで（until）実行される。

■複数の文を実行したい場合，**repeat..until**では単にセミコロンで区切って並べればよいが，do..whileではブロック文を使う。

2つの違いを示す例をあげましょう。

```
repeat                                    do {
  Write('Enter a value: ');                  printf("Enter a value: ");
  Readln(A)                                  scanf("%d",&a);
until (Low <= A) and (A <= High);         } while (a < low || a > high);
```

これを見ると，CとPascalのもう1つの重要な違いに気がつきます。Cでは，関係演算子(>，<，など)は論理演算子(&&，||)より優先順位が高いことです。このため，Pascalではカッコで囲まなければならなかった箇所でも，カッコを省略することができることになります。

for ループ

for ループは，PascalとCでもっとも大きく異なっています。Pascalの**for**ループはかなり固定的で自由度が低いものですが，Cでは非常にフレキシブルで，似ても似つかないものになってしまうこともあります。

各々の書式は次の通りです。

```
for 制御変数 := 初期値 to 最終値 do        for (式1; 式2; 式3)
    文;                                        文;
```

Cでは(実際にはPascalでも)，for文はwhile文の特殊な場合であるにすぎません。for文をwhile文で表わすと次のようになります。

```
式1;
while (式2) {
     文;
     式3;
}
```

式1は初期化，**式2**はループ終了の判定，**式3**はループ変数（複数個も可）のインクリメントあるいは変更に使用されます。

例をいくつか示します。Pascalでは**while**ループも使っています。

Turbo Pascal	Turbo C
`for I := 1 to 10 do begin` `  Write('I = ',I:2);` `  Write('  I*I = ',(I*I):4);` `  Writeln('  I**3 = ',(I*I*I):6)` `end;`	`for (i = 1; i <= 10; i++) {` `   printf("i = %2d  ",i);` `   printf("i*i = %4d  ",i*i);` `   printf("i**3 = %6d¥n",i*i*i);` `}`
`I := 17; K := I;` `while (I > -450) do` `begin` `  K := K + I;` `  Writeln('K = ',K,' I = ',I);` `  I := I - 15` `end;`	`for (i = 17, k = i; i > -450; i -= 15)` `{` `   k += i;` `   printf("k = %d i = %d¥n",k,i);` `}`
`X := D/2.0;` `While (Ads(X*X-D) > 0.01) d` `  X := (X + D/X)/2.0;`	`for (x = d/2; fabs(x*x-d) > 0.01;` `     x = (x+d/x)/2)` `   ;              /* 空の文 */`

3つのループを見ていくと，for 文のカッコ（ヘッダ）の中で行なわれる処理がだんだん多くなっていくのがわかります。最後の例に至っては，必要な仕事はすべてヘッダの中で行なわれ，ループとして実行する文はなくなってしまっています。このような違いは，for ループそのものの違いというよりも，Pascal と C の式についての考え方（カンマ演算子なども含めて）の相違によるものと言えるでしょう。

サブルーチン

Pascal も C もサブルーチンを持っています。Pascal では手続きと関数がありますが，C では関数だけです。しかし，関数を void 型と宣言して，関数が値を返さないことを指定することができます。また，関数が返す値を無視することもできます。

両言語の関数の書式は次のようになります。

Turbo Pascal	Turbo C
<pre>function 関数名 (引数宣言) : 型; ローカルな宣言 begin 文; 文; ... 文 end;</pre>	<pre>型 関数名 (引数宣言) { ローカルな宣言 文; 文; ... 文; }</pre>

Pascal では，引数宣言は同じ型の一組の引数ごとに「**引数, 引数, ... : 型;**」という形式をとります。一方 C では「**型 引数, 型 引数, ...**」となります。

他にも重要な違いはありますが，例を見たほうがよくわかるでしょう。

Turbo Pascal	Turbo C
<pre>function Max(A,B : Integer; begin if A > B then Max := A else Max := B end;</pre>	<pre>int max(int a, int b) { if (a > b) return(a); else return(b); }</pre>

このように，Pascal では関数名への代入によって値を返すのに対し，C では return 文で関数値を返すことに注意してください。

Turbo Pascal	Turbo C
<pre>procedure Swap(var X,Y : Real); var temp : Real; begin temp := X; X := Y; Y := Temp end;</pre>	<pre>void swap(float *x, float *y) { float temp; temp = *x; *x = *y; *y = temp; }</pre>

Pascalでは2種類の引数があります。**変数引数**（アドレスを渡す）と**値引数**（値を渡す）です。Cでは値しか渡せないので，アドレスを渡したい場合は自分でやらなければなりません。つまり，実引数としてはアドレスを渡し，対応する仮引数はポインタと定義するわけです。これが上の *swap* で行なわれていることです。このルーチンを呼び出すコードは以下のようになります。

Turbo Pascal	Turbo C
<pre>Q := 7.5; R := 9.2; Writeln('Q = ',Q:5:1,' R = ',R:5:1); Swap(Q,R); Writeln('Q = ',Q:5:1,' R = ',R:5:1);</pre>	<pre>q = 7.5; r = 9.2; printf("q = %5.1f r = %5.1f¥n",q,r); swap(&q,&r); printf("q = %5.1f r = %5.1f¥n",q,r);</pre>

Cでは，アドレスof演算子（&）を使って，*q* と *r* のアドレスを *swap* に渡しています。

関数プロトタイプ

Pascal と C には，関数について重要な違いが1つあります。Pascal では，関数が呼び出されたときの実引数の個数と型が，宣言された仮引数の個数と型に一致しているかどうかのエラーチェックを必らず行ないます。

たとえば，Pascal で次のような関数を定義したとしします。

```
function Max(I,J : Integer) : Integer;
```

これを，次のように実数値を実引数として呼び出したらどうなるでしょうか。

```
A := Max(B,3.52);
```

“型が一致しない” (type mismatch) というコンパイルエラーが出るはずです。3.52は整数としては受け入れられないからです。

C ではこうなりません。C は，デフォルトでは関数呼び出し時のエラーチェックは行ないません。引数の個数，引数の型，さらには関数が返す値の型もチェックされません。このため定義する前に関数を呼び出すことができるので，柔軟性はより大きくなるわけです。しかし，同時にわかりにくいトラブルにおちいる危険性もまた高くなります（この章の後ろの方で紹介する「**落し穴# 2**」を見てください)。さて，これを避けるにはどうすればよいでしょうか。

Turbo C は，**関数プロトタイプ**をサポートしています。Pascal の **forward** 宣言と似たようなものと言えばわかりやすいかもしれません。関数プロトタイプは，実際にその関数を呼び出す前に，通常はソースファイルの先頭あたりに置きます。キーポイントは，関数プロトタイプ（一種の宣言）は，関数に対する呼び出しより前に置かなくてはならないということです。

関数プロトタイプは，次のような書式になります。

```
型 関数名 (型 引数名, 型 引数名, etc.);
```

Pascal で関数を宣言するときと似ていますが，違いもいくつかあります。まず，各引数の定義の区切りには（セミコロンではなく）カンマを使います。また，1つの型に対して複数の引数名を指定することはできません。次に関数プロトタイプの例を示しますが，使っている関数はこれまでに出てきたものと，この後の「**少し大きなプログラム例**」で出てくるものです。

```
int    max(int a, int b,);
void   swap(float *x float *y);
void   swapitem(listitem *l, listitem *j);
void   sortlist(list l, int c);
void   dumplist(list l, int c);
```

Pascalの**forward**宣言とは異なり，関数プロトタイプがあってもなくても，実際の関数の定義はまったく同じように行ないます（モダンなCのスタイルで定義することもできます)。関数の定義がプロトタイプと一致していなければ，Turbo Cがコンパイルエラーを出してくれます。

Turbo Cは，古典とモダンの両方のスタイルをサポートしています。ただしC言語自体がモダンスタイルに向かって移行しつつあるので，関数プロトタイプと，プロトタイプ形式の関数定義（モダンスタイル）を使用することをお薦めします。

関数プロトタイプを使うことによって，たくさんの問題を防ぐことができます。Cルーチンのライブラリをコンパイルするときなどは特にそうです。このような場合，ヘッダファイルを1つ作って，そのライブラリに含まれるルーチンすべての関数プロトタイプを入れておくとよいでしょう。そして，ライブラリ内のルーチンを使う場合は，プログラムの中でそのヘッダファイルを（#includeを使って）インクルードします。こうすればコンパイル時にエラーチェックが行なわれるので，苦しむ時間を少なくすることができます。

少し大きなプログラム例

さてここで，少し大きなプログラム例を見てみましょう。これは完全なプログラムで，今までに見てきた内容が含まれています。配列 *myList* は，*Item* 型(ここでは単に integer)の要素を持ち，要素の個数は定数 *LMax* で定義されています。プログラムは，最初に配列 *myList* を乱数で初期化し，*DumpList* ルーチンを使ってそれを表示し，*SortList* で昇順にソートして，最後にまた配列の内容を表示します。

ここに示したプログラムのＣバージョンはけっして最良のものとはいえません。Pascal バージョンにできるだけ対応するように作ったためです。また対応しない箇所は，Ｃと Pascal の違いを示すように作成されています。

Turbo Pascal

```
program DoSort;
const
  LMax = 100;
type
  Item = integer;
  List = array[1..LMax] of Item;
var
  myList  : List;
  Count,I : integer;
  Ch      : char;

procedure SortList(var L : List;
                    C : integer);
var
  Top,Min,K : integer;

  procedure SwapItem(var I,J:Item);
  var
    Temp : Item;
  begin
    Temp := I; I := J; J := Temp
  end; { SwapItem 終わり }


begin { SortList のボディ }

  for Top := 1 to C-1 do begin
    Min := Top;
    for K := Top + 1 to C do
      if L[K] < L[Min]
        then Min := K;
```

Turbo C

```
#define LMAX 100

typedef int  item;
typedef item list[LMAX];

list   myList;
int    count, i;
char   ch;

void swapitem(item *1, item *j)
{
  item  temp;

  temp = *i; *i = *j; *j = temp;
} /* swapitem 終わり */

void sortlist(list l, int c)
{
  int top, min, k;
  for (top = 0; top < c-1; top++) {
    min = top;
    for (k = top + 1; k < c; k++)
      if (l[k] > l[min])
        min = k;
```

```
    SwapItem(L[Top],L[Min])
  end
end; { SortList 終わり }

procedure DumpList(L : List;
                   C : integer)
var
  I : integer;
begin
  for I := 1 to C do
    Writelin('L[',I:3,']',L[I]:4)
end; { DumpList 終わり }


begin { DoSort メインボディ }
  for I := 1 to LMax do
    myList[I] := Random(1000);
  Count := LMax;
  DumpList(myList,Count);
  Ch := ReadKey;
  SortList(myList,Count);
  DumpList(myList,Count);
  Ch := ReadKey;

end { DoSort 終わり }
```

```
    swapitem(&l[top],&l[min]);
  }
} /* sortlist 終わり */

void dumplist(list l, int c)

{
  int    i;

  for (i = 0; i < c; i++)
    printf("l[%3d] = %4d¥n",i,l[i]);
} /* dumplist() 終わり */

main()
{
  for (i = 0; i < LMAX; i++)
    myList[i] = rand() % 1000;
  count = LMAX;
  dumplist(myList,count);
  ch = getch();
  sortlist(myList,count);
  dumplist(myList,count);
  ch = getch();

} /* main 終わり */
```

ここで注意すべき点がいくつかあります。

■ Pascal では，手続き *SwapItem* は，手続き *SortList* の中にネスト（入れ子）されていますが，C ではネストにすることはできません。*swapitem* は *sortlist* とは独立して定義されています。

■ C では，配列の添字は常に0から始まり，サイズ-1までになります。したがって配列 *myList* の最初の要素は *myList*[0]，最後の要素は *myList*[*LMAX*-1]になります。この点を考慮して，for ループは配列を正しく処理するように組まれています。

■ *sortlist* に *myList*[]を渡すときには，アドレス of 演算子（&）を使う必要はありません。これは C では，引数として渡されるのは常に配列のアドレスであり，配列がそっくり渡されるわけではないためです。配列すべての値を渡すのはなかなかやっかいだからです。同様に，*dumplist* と *sortlist* の仮引数は list l;と宣言しています。typedef 宣言によって，*list* という型は配列であるとわかるので，ポインタ演算子を使う必要はありません。

■この例では各関数は使用する前に定義してあるので，関数プロトタイプを使用する必要はありませんでした。指定する場合，データ型（*item*, *list*）の定義の後ろならばどこに置いてもかまいません。関数プロトタイプは次のようになります。

```
void swapitem(item *i, item *j);
void sortlist(list l, int c);
void dumplist(list l, int c);
```

関数プロトタイプを使っても関数の定義には影響がないことをもう一度言っておきます。

データ構造についての概要

ここでは，Cで使用できるデータ構造がTurbo Pascalのデータ構造と，どのように似ているのか（あるいは似ていないのか）について述べます。採り上げるのは，ポインタ，配列，文字列，構造体，共用体です。

ポインタ

ポインタはPascalでも使用できますが，Cのようには使われてきませんでした。なぜでしょうか。前にもふれたように，Cでは関数を呼び出す場合，引数としては値しか渡せません。そこで，仮引数に変更を加えて対応する実引数を変更したい場合は，アドレスを渡して，対応する仮引数は実引数のデータ型を指すポインタとして宣言しなければならないのです。さらにCでは，文字列はcharを指すポインタとして実現されているので，文字列を操作しようとすれば，どうしてもポインタを使わざるを得ないのです。

次にPascalとCにおけるポインタの宣言と使用の比較を示します。

	Turbo Pascal	Turbo C
（宣言）	`ポインタ名 : ^型;`	`型 *ポインタ名;`
	`IntPtr : ^Integer;`	`int *intptr;`
	`Buff1 : ^intarray;`	`int buff1[];`
	`Buff2 : array[0..N] of IntPtr;`	`int *buff2[];`
	`PHead : ^Node;`	`node *phead;`
	`Head : Node;`	`node head;`
（使用）	`ポインタ名^ := 値;`	`*ポインタ名 = 値;`
	`IntPtr^ := 22;`	`*intptr = 22;`
	`Buff1^[152] := 0;`	`buff1[152] = 0;`
	`PHead^.Next := nil;`	`(*phead).next = NULL;`
		`/* または phead->next = NULL; */`

Cの例の最後（*phead*）でカッコを使用していること，また*phead*の別の使い方で特殊な記号（->）が使われていることに注意してください。もう少し例を見てみましょう。

```
1 .  Buff1^[152] := 0;          buff1[152]    = 0;
2 .  Buff2[152]^ := 0;          *buff2[152]   = 0;
3 .  Head.Data^  := 0;          *head.data    = 0;
4 .  Head.Next   := nil;        head.next     = NULL;
5 .  PHead^.Next := nil;        (*phead).next = NULL;
                                /* または phead -> next = NULL; */
```

1. *buff1* は整数の配列を指していることが仮定されています。
2. *buff2* は整数を指すポインタの配列であり，これを参照するには添字をつけなければなりません。
3. *head* は **record**(C では **struct**) を指すポインタであり，レコードはフィールド *data* を持ち，それは整数を指すポインタになっています。
4. *head* はフィールド *next* を持ち，それは何かを指すポインタになっています(何を指すかはこれだけではわかりません)。
5. *phead* は **record** (**struct**) を指すポインタであり，レコードはフィールド *next* を持っており，それはポインタになっています。

—>というシンボルは略記法で，たとえば次の式は，

```
pname->fname = value;
```

pname はなんらかの構造体（レコード）を指すポインタであり，*fname* はその構造体のフィールド名で，*pname* が指す構造体のフィールド *fname* に *value* が代入されることを意味します。

配列

Cにおける配列は，Pascal とくらべてかなり簡単なものになっています。配列の添字としては，C では整数・文字・列挙型が許されているだけですが，Pascal では順序づけが可能な型ならどんなものでも使用することができます。また，C では配列の添字は必らず0から始まり，最後は *n*-1（*n* は配列のサイズ）になります。Pascal はこれとは異なり，添字の上限と下限は自由に選ぶことができます。C の配列の添字づけはポインタ演算と似ており，Pascal で $a[i]$ と表現するものは，C では $a[i]$ だけでなく $*(a+i)$ と表現することもできます。

両言語の配列の一般的書式は次のようになります。

```
名前 : array [下限 .. 上限] of 型;          型 名前[サイズ];
```

サイズは（1＋**上限**-**下限**）に等しくなります。

Cの多次元配列は，Pascal と似た方法で宣言することができます。型それ自体が何かの配列であってもよく，最後の方にさらにサイズを指定することもできます。

```
型　名前 [サイズ1] [サイズ2] [サイズ3];
```

Pascal とは異なり，*arr*[*x*][*y*]を *arr*[*x,y*]と書くことはできません（「**落し穴#5**」を参照してください）。

Cでは，指定の型のデータをサイズの個数分格納するのに充分な大きさのメモリブロックが確保され，名前はそのブロックの先頭を指す定数ポインタになります。

これは，関数へ配列を渡す場合に有用になります。Pascal とは異なり，関数側ではコンパイル時に配列の大きさがわかっていなくてもよいわけです。したがって，その関数には（型が同じであれば）大きさの違う配列を渡すことができます。

たとえば，次の関数を考えてみましょう。この関数は int 型の配列を引数にとり，その中の最小値を返します。

```
int  amin(int a[], int n)      /* 関数の定義 */
{
    int  min,i;

    min = a[0];

    for (i = 1; i < n; i++)
        if (a[i] < min)
            min = a[i];
    return(min);
}
```

この関数には，どんな大きさの整数型配列でも渡すことができ，先頭から *n* 個の要素の中の最小値を返します。Pascal ではこれができないので，Cプログラマは不満を感じるわけです。

文字列

標準 Pascal は，文字列を独立したデータ型としては定義していませんが，Turbo Pascal では定義しており，文字列を取り扱う手続きや関数もいくつか用意されています。Cは（Turbo C も含めて），文字列を独立したデータ型として定義していませんが，文字列は，char 型の配列，あるいは char を指すポインタとして定義することができます。

両者の宣言を比較してみましょう。

Turbo Pascal	Turbo C
`名前 : string[サイズ];`	`char    名前[サイズ];`
`type`	
`  BigStr  = string;`	`typedef char bigstr[256];`
`  StrPtr  = ^BigStr;`	`typedef char *strptr;`
`var`	
`  Line      : string[80];`	`char    line[81];`
`  Buffer    : BigStr;`	`bigstr  buffer;`
`  word      : string[35];`	`char    word[36];`
`  Ptr       : StrPtr;`	`strptr  ptr;`

Turbo Pascal と Turbo C それぞれの文字列の重要な違いは，両言語の配列の違いに起因しています。

Turbo Pascal では，次の2つの宣言はまったく等価です。

```
S : string[N];
S : array of [0..N] of char;
```

文字列の最大長は N 文字で，現在の長さは $S[0]$ に格納されています。実際の文字列は $S[1]$ から始まります。文字列リテラルや文字列定数は，文字列変数の中に直接代入することができます。Pascal では，バイトごとの転送を行なって長さを正しく調整します。

Turbo C では，文字列の宣言には次の2つの方法があります。

```
char strarr[N];     /* 文字配列        */
char *strptr;       /* 文字へのポインタ */
```

前者の場合，文字列を格納するための N バイトが確保され，***strarr*** はそのメモリ領域の先頭アドレスを表わします。後者は，***strptr***（char 型を指すポインタ）を格納する数バイトが確保されるだけです。

C では，文字列の長さはどこにも記憶されていません。そのかわり，文字列の最後に**文字列終了子**というマークをつけます。終了子は**ヌル文字**（ASCII コードの0）で，文字列の終わりに1バイトが付加されるわけです。

したがって ***strarr*** の中には，実際の文字は *N-1* 個しか入らないことになります。最後の1バイトはヌル終了子のためにとられるからです。上の比較表で，C の宣言はすべて Pascal

の宣言より1大きくなっているのはこのためです。

strarr は，実際にバイトが集まってできているわけではないので，これに文字列を直接代入することはできません。**strcpy** ルーチン（あるいは同等な関数）を使って，ある文字列から他の文字列へバイトごとの転送を行なってやらなければなりません。

```
strcpy(strarr, "hello, world!");
```

しかし，**scanf** や **gets** を使えば，*strarr* に文字列を直接読み込むことができます。

もう1つの方法（char * strptr）で文字列を宣言する場合は，もっと注意する必要があります。*strptr* は char を指すポインタであり，ポインタ自身を格納する数バイトが確保されているだけで，文字列を収容するためのメモリ領域はなんら確保されていません。

この場合，文字列リテラルを *strptr* に直接代入することはできます。文字列リテラルはオブジェクトコードの中の一部分として作り出され，そのアドレスを *strptr* に代入すればよいからです。*strarr* を *strptr* に代入すると，*strarr* と *strptr* は同じ文字列を指すことになります。他の文字列を指すポインタを strptr に代入しても，同じようなことが起こります。

では他の文字列ではなく，*strptr* 自身が独自の文字列を指すようにするにはどうすればよいでしょうか。これには，*strptr* に領域を割り当てなければなりません。

```
strptr = (char*) malloc (N);
```

malloc 関数によって *N* バイトのメモリが確保され，その先頭アドレスが *strptr* に代入されます。こうすれば，**strcpy** を使って文字列（リテラルでも変数でも）を確保した領域にコピーすることができます。

先程の例で，Pascal でこれに対応しているもの（StrPtr,Ptr）は，完全に同じ機能を実現しているわけではありません。*strptr* は， *char* ではなく， *BigStr* と定義されています。これは Turbo Pascal が，*Ptr* は文字列であると認識できるようにするためです。こうすると範囲チェックに関する問題も避けることができます。次の例では，実際に *Ptr* に割り当てられる領域は，要求された分だけであることに注意してください。

次に示すのは，一応対照できる例です。Turbo C の文字列操作関数（**str...**）についてはリファレンスガイドを参照してください。ここでは，先程の例の型宣言が行なわれているものと考えてください。

Turbo Pascal	Turbo C

```
var
  Line,Name    : string;
  First,Temp   : string[80];
  Ptr          : StrPtr;
  I,Len,Err    : integer;

begin
  Write('Enter name:  ');
  Readln(Name);
  I := Pos(' ',Name);
  if I = 0 then
    First := Name
  else
    First := Copy(Name,1,I-1);
  Len := Length(First);
  Writeln('Len = ',Len);
  Temp := Concat('Hi, ',Name);

  Writeln(Temp);
  if Name <> First
    then Name := First;
  I := 823; Str(I,Temp);
  Val(Temp,I,Err);
  GetMen(Ptr,81);
  Ptr^ := 'This is a test.';
  Writeln('Ptr = ',Ptr^);
  FreeMen(Ptr,81)
end.
```

```
main()
{
  bigstr line,name
  char   first[81],temp[81];
  char   *ptr;
  int    i,len;
  extern char *strchr(char *s, char ch);

  printf("Enter name:  ");
  gets(name);
  ptr = strchr(name,' ');
  if (ptr == NULL)
     strcpy(first,name);
  else
     strncpy(first,name,ptr-name-1);
  len = strlen(first);
  printf("len = %d¥n",len);
  strcpy(temp,"Hi, ");
  strcat(temp,name);
  puts(temp);
  If (strcmp(name,first))
     strcpy(name,first);
  i = 823; sprintf(temp,"%d",i);
  i = atoi(temp);
  ptr = (char *) malloc(81);
  strcpy(ptr,"This is a test.");
  printf("ptr = %s¥n",ptr);
  free(ptr);
}
```

Pascal のソースコードでの *Ptr* の使い方が，C ではどんなものに対応するのかを示すために作り出したものなので，やや不自然なところもあります。

補足：例の中で呼び出している C のルーチンに対する関数プロトタイプは，ヘッダファイル（.h）の中に入っています。したがってエラーチェックを行なうためには，プログラムの先頭に次のような# include 文を置く必要があります。

```
#include  <stdio.h>
#include  <string.h>
#include  <stdlib.h>
#include  <alloc.h>
```

構造体

Pascal と C どちらの言語も，さまざまな型のデータからなるデータ構造をサポートしています。Pascal では**レコード**，C では**構造体**と呼ばれます。この2つの書式を示しましょう。

Turbo Pascal	Turbo C
<pre>type レコード型名 = record フィールド名 : 型; フィールド名 : 型; ... フィールド名 : 型 end; var 変数名リスト : レコード型名;</pre>	<pre>typedef struct { 型 フィールド名; 型 フィールド名; ... 型 フィールド名; } 構造体型名; 構造体型名 変数名リスト;</pre>

C には，もっと簡単に構造体型変数を直接宣言する方法があります。もちろん Pascal にもあります。

Turbo Pascal	Turbo C
<pre>var 変数名 : record フィールド名 : 型; ... フィールド名 : 型 end;</pre>	<pre> struct 構造体型名 { 型 フィールド名; ... 型 フィールド名; } 変数名;</pre>

Cの構造体型名はオプションで，指定しておけば他の変数を同じ型で宣言したい場合にも使えます。それ以外には，Pascalのレコードと Cの構造体は互いによく似ています。次に例を示します。

Turbo Pascal	Turbo C
<pre>type Student = record Last,First : string[20]; SSN : string[11]; Age : Integer; Tests : array[1..5] of Integer; GPA : Real end; var Current : Student; begin Current.Last = 'Smith'; Current.Age = 21; Current.Test[1] = 97; Current.GPA = 3.94; end.</pre>	<pre>struct student { char last[20],first[20]; char ssn[11]; int age; int tests[5]; float gpa; } current ; main() { strcpy(current.last,"Smith"); current.age = 21; current.test[0] = 97; current.gpa = 3.94; }</pre>

ここで，Pascal と Cの唯一の大きな違いは，Pascalは **with** 文を持っているのに対し，Cにはないということです。上のPascalのコードを **with** *Current* **do** を使って書きかえれば，*Current.*をつけなくても直接フィールドを参照することができます。Cでは，必らず先頭に *Current.*をつけなくてはなりません。なおCには，もう1つ**メンバアクセス演算子**があります。これは演算子の左の識別名が，構造体そのものではなく，構造体を指すポインタの場合に使用されます。たとえば，*pstudent* が構造体を指すポインタだとすると，次の文は，Jones という文字列を *last* というメンバに代入します。

```
pstudent -> last = "Jones";
```

共用体

Pascal と C はもう1つ同じような概念をサポートしています。Pascal では**可変レコード**，C では**共用体**と呼ばれているものです。各々の定義を示しましょう。

Turbo Pascal	Turbo C
<pre>type 共用体型名 = record フィールドリスト case 型 of ケース : (フィールドリスト); ケース : (フィールドリスト); ... ケース : (フィールドリスト) end;</pre>	<pre>union 共用体型名 { 型 フィールド名リスト; 型 フィールド名リスト; ... 型 フィールド名リスト; };</pre>

Pascal でフィールドリストというのは，通常「**フィールド名：型；**」の並びです。

Pascal と C の間には次の2つの大きな違いがあります。

まず，Pascal では，通常のレコードの最後に可変部（共用体）を置くことができますが，C ではそれができません。ただし，先に共用体を宣言しておいて，構造体の中にその共用体型のフィールドを置くことはできます。

第2に，Pascal では，各可変部にさまざまな型を指定することができます。C では，複数のフィールドを指定することはできますが，その型はすべて同じでなければなりません。

例をあげましょう。Pascal と C は，なるべく似せて書いてあります(まったく同じとはいきませんが)。

Turbo Pascal	Turbo C
<pre>type trick_word = record case integer of 0 : (w : integer); 1 : (lob,hib: byte); end; var xp : trick_word;</pre>	<pre>typedef union { int w; struct { char lob; char hib; } b; } trick_word; trick_word xc;</pre>

C と Pascal 両方とも，*trick _word* の定義は可搬性のあるものではないことに注意しておいてください。どちらも8086のバイトの並び方に依存しています。

C の共用体では，構造体と同様に，右カッコ'}'とセミコロン'；'の間に変数名のリストを指定することができます。こうすると，変数の宣言が直接行なえます。この場合，同じ共用体型の変数を他に宣言するつもりがなければ，共用体型名は指定しなくてもかまいません。Pascal では，フィールドを参照するには，*xp.w*，*xp.hib*，*xp.lob* を使うことになりますが，C ではそれぞれ *xc.w*，*xc.b.hib*，*xc.b.lob* になります。

プログラミング上の問題点

Pascal を知っていれば，Turbo C に慣れるのにそれほど問題はなかったことと思います。しかし，両言語で少し実現のしかたが異なる領域がいくつかあります。ここでは，そうした点について検討を加えていくことにします。

大文字/小文字の区別

Pascal では大文字/小文字の区別を行ないませんが，C はこれを区別します。したがって，indx，Indx，INDX は，Pascal ではすべて同じ変数を示しますが，C ではまったく別の変数になります。

注意：関数の呼び出しは，C のプログラムがリンクされるまでは解決されないので，大文字小文字の区別に基づく違いはリンク時まで明らかにはなりません。したがって，C ではくれぐれも大文字小文字の区別に関しては気をつけてください。

型の変換（型キャスト）

Pascal では原則的に，限られた範囲でしか型のキャスト（データを別の型に変換する）は行なうことができません。関数 ***Ord***() は順序づけ可能な型から整数型へ，***Chr***() は整数型（あるいはその関連型）から文字型への変換を行ないます。Turbo Pascal では，型の変換がさらにいくつか許されており，順序づけ可能な型(整数型，論理型，文字型，列挙型)の間での型変換(リタイプと呼ばれている)ができます。C はもっと自由で，どんな型からどんな型へでも型変換できます。ただし，いつも望みどおりの結果が得られるとは限りませんが。

それぞれの標準的な書式と，その例を示します。

Turbo Pascal	Turbo C
変数 := 型(式);	**変数 = (型) 式;**
var Ch : Char;	char ch;
I := integer(Ch);	i = (int) ch;
Ch := char(Today);	ch = (char) today;
Today := Days(3);	today = (days) 3;

これに加えて，Turbo C では，自動的に型のキャストが行なわれる場合があります。**整数コンパチブル**（内部表現が整数値で表わせるもの）の値の間での場合です。そのため，上の3つの代入文は，明示的なキャストをやめて，次のように書くことができます。

```
i = ch;
ch = today;
today = 3;
```

定数, 変数の記憶クラス, 初期化

Turbo Pascal では，宣言した変数は初期化されません。また，手続きおよび関数の内部で宣言された変数の値は，呼び出しと呼び出しの間では保持されません。唯一の例外は型つき定数で，初期化が行なわれ，その定数が定義されているサブルーチンの呼び出しの間も値は保持されます。C では，すべてのグローバル変数は，デフォルトでは0に初期化され，指定があればその値に初期化されます。

注意：初期化されない変数は，必らずしも宣言された順序で作り出される必要はないということを憶えておいてください。

Turbo C には2種類の定数があり，変数の初期化が可能であり，また関数の中で変数を static と宣言することもできます。

定数型

次に2種類の定数の定義の書式を示します。

```
#define  定数名  値;
const  型  定数名 = 値;
```

最初の定義 (# define...) は，Pascal の **const** 定義によく似ています。定数名が見つかった箇所は，すべて値で書きかえられるからです。

2番目 (const...) は，定数名 (の値) が変更できない点を除いて，Turbo Pascal の型つき定数に似ています。定数名を変更しようとすると，コンパイルエラーになります。

C では，次のような定数の式が許されていることを憶えておいてください。

```
char s[SIZE + 1];
```

const 型の変数はこのように使うことはできません。はっきりとした定数 (# define...) はこれが可能です。ただし，その置き換えはワードプロセッサのように行なわれるので，思わぬ結果になってしまうことがあります。

```
#define MAX_SIZE  80 + 1
char    s[MAX_SIZE * 2];    /* 162ではなく82になる */
```

思いどおりの結果を得るためには，次のように定義しなければなりません。

```
#define MAX_SIZE  (80 + 1)
```

つまり，置きかえを正しく行なわせるためには，# define 文の式はカッコで囲んでおく必要があるわけです。

変数の初期化

Turbo C では，Turbo Pascal の型つき定数と同じように，変数を初期化することができます。その書式は次のようになります。

```
型  変数名 = 値;
```

複数の値を代入する必要がある場合（配列，構造体）は，値をカンマで区切って，全体を中カッコ{}で囲みます（{"like_this", "and_this", "and_this_too"}）。

```
int x = 1, y = 2;
char name[] = "Frank";
char answer = 'Y';
char key = 3;
char list[2][10] = { "First", "Second" };
```

変数の記憶クラス

C では，いくつかの変数の記憶クラスが定義されています。この中で重要なのは，**外部変数**と**自動変数**（ローカル変数）の2つです。グローバル変数（**main** も含めて関数の外側で宣言される変数）はデフォルトで外部変数となります。つまり，初期値を指定しない限り，プログラムの実行開始時に0に設定されます。

関数（**main** も含む）の内部で宣言された変数は，デフォルトで自動変数となります。初期値を指定しなければ初期化は行なわれず，その関数の呼び出しと呼び出しの間では，その値は消えてしまいます。しかし，変数を **static** と宣言することができ，その場合は（プログラムの実行開始の際に1回）0に初期化され，呼び出しと呼び出しの間でも値は保持されます。

次の例を見てください。

```
int test(void)
{
    int i;
    static int count;
    ...
}
```

変数 *i* はスタック上にとられ（自動変数），関数 *test* が呼び出されるたびに，関数の中で値を設定しなければなりません。一方静的変数 *count* は，グローバルデータ領域に格納され，プログラムの実行が開始されたときに0に初期化されます。*count* は，*test* が呼び出されるときには，以前の値を保持していることになります。

動的メモリ割り当て

Turbo Pascal では，ヒープを管理する方法はいくつかあります。Turbo Pascal で次のような宣言を行なったとしましょう。

```
type
  ItemType = Integer;
  ItemPtr = ^ItemType;
var
  p : ItemPtr;
```

動的にメモリの割り当てと解放を行なう方法は3つあります。

```
(* New and Dispose *)

New(p);                     { 必要なだけの領域を自動的に割り当てる   }
...
Dispose(p);                 { 割り当てられた領域を自動的に解放する   }

(* New, Mark, Release *)

New(p);                     { 必要な分の領域を自動的に割り当てる     }
...
Mark(p);
Release(p);                 { ^p からヒープの最後までを解放する      }

(* FreeMem, GetMem *)

GetMem(p, SizeOf(ItemType));   { 割り当てる領域の大きさを指定する }
...
FreeMem(p, SizeOf(ItemType));  { 解放する領域の大きさを指定する   }
```

Turbo C では，メモリの動的な割り当てと解放には，Turbo Pascal の ***GetMem*** と ***Dispose*** に似たルーチンを使うことになります。

```
型 * ポインタ;
ポインタ = (型 *) calloc (個数 , 大きさ);
/* または  ポインタ = (型 *) malloc (必要となるバイト数);          */
/* または  ポインタ = (型 *) realloc (ポインタ, 必要となるバイト数); */
free (ポインタ)

typedef int ItemType;
Itemtype *p;

p = (ItemType*) malloc(sizeof(ItemType));
...
free(o);
```

この3つのCルーチン（**calloc**, **malloc**, **realloc**）はすべて，汎用のポインタ，つまりどんな型にもキャストできるポインタを返します。またヒープ領域に必要なメモリが無い場合はどの関数もNULLを返します。

- **calloc** には，確保したい項目の数と項目1個のサイズ（バイト単位）を指定します。**calloc**は，指定された個数の項目を作ってすべて0で埋め，このブロックを指すポインタを返します。これは動的に配列を作り出すときによく使用されます。
- **malloc** には確保したいバイト数を指定します。
- **free** は，引数に指定したポインタが指しているメモリを解放します。

コマンドライン引数

Turbo Pascal で.EXEファイルを作ると，そのプログラムはコマンドラインで入力した引数を読み込むことができます。これには ***ParamCount*** と ***ParamStr*** を使用します。たとえば，DUMPIT.EXEというプログラムを作って，次のようなコマンドラインで起動したとすると，

```
A>dumpit myfile.txt other.txt 72
```

ParamCount は3になり，***Paramstr*** は次のような文字列を返します。

```
ParamStr(1)     myfile.txt
ParamStr(2)     other.txt
ParamStr(3)     72
```

同様にTurbo Cでも（標準のCのやり方にならって），**main** の引数として識別名 ***argc***, ***argv***, ***env*** を次のように宣言することができます。

```
main(int argc, char *argv[], char *env[])
{
   /*  mainのボディ  */
}
```

argc は引数の個数，***argv***[]は引数を保持している文字列の配列です。上と同じコマンドラインで考えると，***argc*** は4，***argv***[]は次のようになります。

```
argv[0]     A:¥DUMP.EXE
argv[1]     myfile.txt
argv[2]     other.txt
argv[3]     72
argv[4]     (null)
```

Cでは，MS-DOSバージョン3.xのもとでは ***argv***[0]が定義されており（***ParamStr***(0)は定義されていません），実行されるプログラムの名前を含んでいます。MS-DOSバージョン2.xでは ***argv***[0]はヌル文字列（""）を指しています。なお，***argv***[4]はNULLを持っている（空文字列である）ことに注意してください。

3番目の引数 ***env***[]は文字列の配列で，次のような文字列を保持しています。

環境変数 = 値

右辺の**値**が，左辺の**環境変数**にセットされていることを示しています。

ファイル入出力

標準Pascalでは2種類のファイルが存在します。1つは**テキストファイル**(textと宣言される)，もう1つは**データファイル**（file of 〈型〉と宣言される）です。ファイルのオープン，読み書き，クローズの手順はどちらの場合でもほとんど同じです。Turbo Pascalではもう1つ型のないファイルが用意されています。これはTurbo Cで用いられるバイナリファイルによく似たものです。

Cのファイルは普通どれも連続したバイトの流れ（ストリーム）として扱われます。**fopen**における *t*（**テキスト**）と *b*（**バイナリ**）修飾子は重要ですが，テキストとデータの判別は大部分はユーザにまかされているわけです。

次に2つの言語の間の大まかな対応をいくつか示します。

表 9.2　ファイル入出力の類似点

Turbo Pascal	Turbo C
var	
I : integer;	int i;
X : real;	float x;
Ch : char;	char ch;
Line : string[80];	char line[80];
myRec : RecType;	struct rectype myrec;
buffer : array[1..1024] of char;	char buffer[1024]
F1 : text;	FILE *f1;
F2 : file of RecType;	FILE *f2;
F3 : file;	FILE *f3;
Assign(<fvar>,<fname>); Reset(<fvar>); Reset(<untyped fvar>,<blocksize>);	<fvar> = fopen(<fname>,"r"); /* or <fvar> = fopen(<fname>,"r+"); */ /* or f1 = fopen(<fname>,"r+t"); */ /* or f2 = fopen(<fname>,"r+b"); */
Assign(<fvar>,<fname>); Rewrite(<fvar>); Rewrite(<untyped fvar>,<blocksize>);	<fvar> = fopen(<fname>,"w"); /* or <fvar> = fopen(<fname>,"w+"); */ /* or <fvar> = fopen(<fname>,"w+t"); */ /* or <fvar> = fopen(<fname>,"w+b"); */
Assign(<fvar>),<fname>); Appnd(<text fvar>);	<fvar> = fopen(<fname>,"a+"); /* or <fvar> = fopen(<fname>,"a+t"); */ /* or <fvar> = fopen(<fname>,"a+b"): */
Read(F1,Ch);	ch = getc(f1);
Readln(F1,Line);	fgets(f1,80,line);
Readln(F1,I,X);	fgets(f1,80,line); sscanf(line,"%d %f", &i,&x);
Read(F2,myRec);	fread(&myrec,sizeof(myrec),1,f2);
BrockRead(F3,buffer,SizeOf(buffer));	fread(&buffer,1,sizeof(buffer),f3);
Write(F1,Ch);	fputc(ch,f1); /* または fprintf(f1,"%c",ch); */
Write(F1,Line);	fputs(line,f1); /* または fprintf(f1,"%s,line); */
Write(F1,I,X);	fprintf(f1,"%d %f,i,x);
Writeln(F1,I,X);	fprintf(f1,"%d %f¥n",i,x);
Write(F2,myRec);	fwrite(&myrec,sizeof(myrec),1,f2);
Seek(F2,<rec#>);	fseek(f2,<rec#>*sizeof(rectype),0);
Flush(<fvar>);	fflush(<fvar>);
Close(<fvar>);	fclose(<fvar>);
BlockWrite(F3,buffer,SizeOf(buffer));	fwrite(&buffer,1,sizeof(buffer),f3);

ここで使用した Turbo C のルーチンそれぞれの働きについてはリファレンスガイドを

参照してください。

次に短いプログラムの例を示します。これは，あるテキストファイル（ファイル名はコマンドラインから入力する）を画面上にダンプするプログラムです。

Turbo Pascal

```pascal
program DumpIt;
var
  F  : text;
  Ch : char;
begin
  Assign(F,ParamStr(1));
  {$I-} Reset(F); {$I+}
  if IOResult <> 0 then begin
    Writeln('Cannot open ',ParamStr(1));
    Halt(1)
  end;

  while not EOF(F) do begin
    Read(F,Ch);
    Write(ch)
  end;
  Close(F)
end.
```

Turbo C

```c
#include <stdio.h>
main(int argc, char *argv[])
{
  FILE  *f;
  int   ch;

  f = fopen(argv[1],"r");

  if (f == NULL) {
    printf("Cannot open %s¥n",argv[1]);
    return(1);
  }

  while ((ch = getc(f)) != EOF)

    putchar(ch);

  fclose(f);
}
```

Pascal プログラマがおちいりやすい落し穴

Pascal と C には多くの類似点がありますが，またそのために犯しがちな誤りもいくつかあります。ここではそうした誤りのうち，気をつけなければならないものをいくつか示しています。落し穴の順序は，起こる頻度，Pascal プログラマにとってどの程度見つけにくいか，コンパイラがとらえられるかどうかという点を総合的に見て決めてあります（第7章でもこのようなよく犯す誤りをいくつかあげています）。

落し穴#1：代入と比較

A ＝ B という式は，Pascal では「*A* と *B* が等しい」ことを表わす論理式で，*True* か *False* を返します。一方 C では「*A* に *B* の値を入れる」という代入を意味します。そして重要なのは，この式も値，すなわち *A* に代入された *B* の値を返すということです。Pascal プログラマが最も犯しがちな致命的なバグは次のような文です。

```
if (A = B) 文;
```

C ではこれは完全に文法にかなっており，次のように解釈されます。

- *B* の値を *A* に代入する。
- A ＝ B という式の値は *B* の値となる。
- 式の値が0以外（C では真を示す）であれば文が実行される。

本当に書きたい文は次のようになるでしょう。

```
if (A == B) 文;
```

つまり，*A* と *B* の値が等しい場合に文を実行するということです。

次のことを憶えておきましょう。「等しいかどうかの比較には＝＝を使う。＝ではない。＝は C では代入演算子である。」

落し穴# 2:アドレスを渡すのを忘れる(特にscanfで)

すでに説明したように, Cでは関数への引数の受け渡しは値によってのみ行ないます。つまり, もし引数としてアドレスを受け渡したい場合は, はっきりとそのアドレスを渡す必要があります。この章の初めの方で出てきた *swap* 関数を書いたとしましょう。*swap* を呼び出すときに (変数 *q* と *r* が float 型だとすると), swap(q,r);と書いてはいけません。こう書いてしまうと, *swap* は受けとった *q* と *r* の値をアドレスと考えて, そのアドレスにある2つの値を入れかえてしまいます。

この落し穴はどうやって避ければよいでしょうか。一番よいのは関数プロトタイプを用いることです。そうすれば, コンパイル時に Turbo C がエラーチェックを行なうことができます。*swap* の場合は, ソースファイルの最初の方に次の文を入れておきます。

```
void swap(float *x, float *y);
```

これで, プログラムに swap(q,r);という文があると, コンパイルしたときに *swap* の呼び出しで「引数 *x* の型が宣言と一致しない (Incompatible type conversion)」というエラーメッセージが出力されるはずです。

落し穴# 3:関数呼び出しでカッコをつけ忘れる

Pascal では, 引数を持たない手続きは単に手続名を書いて呼び出します。

```
AnyProcedure;
i := AnyFunction;
```

C の関数呼び出しは, たとえ引数がなくても必らずカッコ()をつけなければなりません。したがって次の呼び出しは誤りです。

```
AnyFunction;          /* 実際には何も起きない                  */
i := AnyFunction;     /* AnyFunction のアドレスが i に代入される */
```

正しくは次のようになります。

```
AnyFunction();        /* AnyFunction を呼び出す                */
i = AnyFunction();    /* AnyFunction を呼び出し, 結果を i に代入する */
```

落し穴#4：警告メッセージ

エラーメッセージの他に，Turbo C は致命的ではない警告を出します。上で示した誤った関数呼び出しに対して，Turbo C は次のような警告を出すはずです。

```
Warning TEST.C 5: Code has no effect in function main
Warning TEST.C 6: Non-portable pointer assignment in function main
```

上の2つの文は文法上は正しいのでエラーにはならず，オブジェクトファイルは作成されるでしょう。しかし注意してください。このような警告は，Turbo Pascal であれば必らず致命的エラーになるものです。Turbo C の警告メッセージを軽視してはいけません。

落し穴#5：多次元配列の添字

matrix という2次元の配列があって，その(i, j)の位置を参照したいとしましょう。Pascal プログラマとしては次のように書きたいところです。

```
x = matrix [i,j];
```

まったく問題なくコンパイルされますが，思うようには動きません。

C では，式をカンマで区切って並べることができますが，こうすると式全体の値は，並びの最後の式の値になります。つまり，上の文は次と同じことになってしまいます。

```
x = matrix [j];
```

意図したものとは異なりますが，C の文としては正しい文です。ただしこの文に対しては警告が出力されます。C は，*matrix*[*j*]（*matrix*[]という行列の *j* 番目の行）のアドレスを *x* に代入しようとしていると認識するからです。

C では配列の添字はそれぞれカッコ[]で囲まなければなりません。したがって次のように書けばよかったのです。

```
x = matrix [i][j];
```

次のことを憶えておきましょう。「多次元配列では，それぞれの添字をカッコで囲まなければならない。」

落し穴#6:文字型配列と文字ポインタの違いを忘れる

次のような文があるとします。

```
char   *str1, str2[30];
str1 = "This is a test";
str2 = "This is another test";
```

最初の代入文は正しいのですが，2番目は正しくありません。なぜでしょう。*str1* は文字列を指すポインタです。コンパイラはこの文を見ると，オブジェクトファイルのどこかにThis is test という文字列を埋め込み，そのアドレスを *str1* に代入します。

一方 *str2* は定数のポインタで，どこかにとられた30バイトのブロックを指しています。*str2* が持っているアドレスを変えることはできません。これは次のように書かなければなりません。

```
strcpy(str2,"This is another test");
```

こうすると，This is another test という文字列から，*str2* が指しているアドレスへ1バイトずつコピーが行なわれます。

落し穴#7：大文字/小文字の区別を忘れる

Pascal では，識別子 *indx*，*Indx*，*INDX* はすべて同じです。大文字と小文字は同じものとして扱われます。しかし C では違います。

```
int Indx;
```

と宣言した後,

```
for(indx = 1; indx<10; indx++) ···· ;
```

と書いてしまうと，コンパイラは *indx* が認識できない（Undefined symbol）というエラーを出力します。

落し穴# 8:ブロックの最後の文にセミコロンを忘れる

本当に必要な箇所以外にはセミコロンは絶対に置かないという Pascal の潔癖症ぎみの人は，しばらくこの問題に悩まされます。さいわいコンパイラはこのエラーを検出してはっきり示してくれるので，それほど深刻な問題にはなりません。C の文は，2つの例外を除いて，最後にセミコロンをつけなくてはならないということを憶えておきましょう。

例外の1番目は関数定義です。

```
型　関数名（引数名並び）
```

ここではカッコのあとにセミコロンをつけてはいけません。ただし関数プロトタイプでは違います。

```
型　関数名（型　引数名，型　引数名，...);
```

これは関数を宣言するもので，定義をするものではありません。Pascal の **forward** 宣言に少し似ています。

2番目の例外は，次のようなプリプロセッサコマンド（#コマンド名）です。

```
#include    <stdio.h>
#define     LMAX 100
```

これを# define LMAX 100;としてしまうと，プリプロセッサは *LMAX* をすべて *100*；に置き換えてしまいます。

C では，とにかく注意深くなければならないことを肝に銘じてください。C は Pascal のように寛容な言語ではないのです。

第10章

Turbo Prologとのインターフェース

Turbo Cの登場によって，パーソナルコンピュータの上で2つの強力な言語を組み合わせて使用することができるようになりました。Turbo CのモジュールとTurbo Prologのモジュールをリンクすれば，Turbo CのアプリケーションにAI（人工知能）的な処理を組み込むことが可能です。Cのプログラミングが得意な人なら，他のC処理系と比較してどんな利点があるかもうおわかりでしょう。またCを学び始めた人もこの章を読めば，2つの言語を組み合わせることによって，各々の言語の機能が拡大されるのがわかると思います。

Turbo Cは手続き型言語であり，一方Turbo Prologは論理プログラミングに基づく言語です。Turbo CのアプリケーションにTurbo Prologをリンクすれば，次のような人工知能的な特徴を得ることができます。

■ルールベースの制御構造
■容易な自然言語処理

また，Turbo Prologとのリンクよって，Turbo CのアプリケーションにAI的機能を付け加えることができるので，複雑な問題を解決するのにも，単に問題を記述して，後はTurbo Prologへのインターフェースエンジンを動かしてやるだけですむわけです。Turbo Cのアプリケーションは多くの場合，Turbo Prologのプログラムとリンクすることによって，開発の時間が短縮され，またコードはわかりやすくなり，プログラムの柔軟性も増します。

この章では……

この章では，Turbo CとTurbo Prologのコンパイルとリンクのステップについて説明しています。また，処理の流れを示すために実例を4つ用意しています。最初の例では，簡単なプログラムをコンパイル・リンクしています。2番目はもう少し複雑で，Cのライブラ

りを一緒にリンクする方法を示しています。3番目では，メモリの割り当てを行ないます。そして最後の例では，実際にグラフィックプログラムを使って，この2つの言語を組み合わせて得られる機能を少しばかり紹介しています。

Turbo C と Turbo Prolog のリンクの概要

Turbo C のモジュールをコンパイルして，Turbo Prolog のモジュールやプログラムとリンクするのはそれほど難しいことではありません。ただし，以下に示すような点には気をつける必要があります。

プログラムモジュールのコンパイル：

- Turbo Prolog から呼び出す C の関数には，Turbo Prolog 側で as "<*filename*>"による拡張をしない場合には，関数名の最後に_0というサフィックス（接尾辞）が必要です（最初の例題の C プログラム CSUM.C を参照)。
- C のメインモジュールのかわりに Turbo Prolog のメインモジュール（ゴールを含むもの）を用意します。
- Turbo Prolog のメインモジュールでは，C の関数をグローバルな述語として定義しなければなりません（最初の例題の Prolog のプログラム PROSUM.PRO を参照)。
- プログラムモジュールはすべてラージメモリモデルでコンパイルします (Prolog にはラージメモリモデルしかありません)。
- プログラムの中で Turbo Prolog バージョン1.1～2.0のライブラリを呼び出す場合は，レジスタ変数を使用しないようにしておく必要があります（-r-)。
- 識別名の下線はつけないようにしておきます（-u-)。

プログラムモジュールのリンク：

- INIT.OBJ をオブジェクトファイルの先頭に置いてリンクします（このファイルは Turbo Prolog の初期化モジュールで，Turbo Prolog のライブラリディスクに入っています)。
- Turbo C のライブラリルーチンが必要な場合は，CL.LIB を使います。また実数演算を行なう場合は，EMU.LIB または MATH.LIB を使います。
  リンクのコマンド行は次の形式です。

```
tlink init cpinit <T_Prolog_Main> Other_files <T_Prolog_Main.sym>,
      [exename],[your_lib] prolog [emu mathl] cl
```

（以上は1行のコマンドなので途中で改行しないでください）

これらの他にも，次のような点に注意してください。

■ Turbo Prolog の関数からは，組み込みの述語と同じようにCで書かれた関数を呼び出すことができます。

■ Turbo Cのライブラリ関数を呼ぶ場合は，必らず先頭に下線（_）をつけてください。

注意：Turbo Cのライブラリ関数はすべて先頭に下線がついています。したがって下線なしでコンパイルしている場合は，ライブラリ関数に対しては明示的に下線をつける必要があります（ユーザ定義の関数には下線は必要ありません)。

■ Turbo Cの **malloc**, **calloc**, **free**, およびその他のメモリ割り当て関数は，Turbo Prolog では **alloc_gstack**, **_malloc**, および **_free** に置き換えられます。この3つは，Turbo C関数の中でメモリ割り当てを行なうために，Turbo Prolog から使用することができます。

alloc_gstack はグローバルスタックにメモリを確保する関数で，次の形式で呼び出します。

```
void *alloc_gstack(unsigned size);
```

_malloc は Prolog のヒープ領域にメモリを確保する関数で，次の形式です。

```
char *_malloc(unsigned size);
```

_free は Prolog のヒープ領域に確保したメモリを解放する関数で，次の形式です。

```
void _free(void *ptr, unsigned size);
```

alloc_gstack が使用される場合には，ユニフィケーションに失敗して割り当てたメモリを通り越してしまうようなバックトラックが起こると，メモリは自動的に解放されます。

■ Turbo C と Turbo Prolog をリンクした場合には，**printf**, **putc** などの画面出力関数は働きません。そのかわりに，**wrch** で Prolog のウィンドウに文字を出力することができます。また，**zwf** は Turbo Prolog の **writef** と同じ機能を持っています。**zwf** は機能の制限された **printf** と考えることができます。

```
zwf(FormatString, Arg1, Arg2, ...)
```

FormatString は，**printf** 形式の書式文字列です。書式指定の詳細については，Turbo Prolog Reference Manual を参照してください。

zwf と **wrch** は，PROLOG.LIB に入っています。

■ Turbo Prolog から呼ばれるCの関数は値を返してはならず，void 型として定義する

必要があります。引数のフローパターンは Turbo Prolog のグローバル述語の宣言で指定します。たとえば次の文は，

```
factorial(integer,real) - (i,o)  language c
```

関数 *factorial* は2つの引数をとり，1番目は整数，2番目は実数（浮動小数点数）であるということを Turbo Prolog に知らせています。(i,o) は，1番目の引数（整数）は値が渡され，2番目の引数は *factorial* の中で代入される浮動小数点数へのポインタであることを示しています。文の最後の c は，この関数は C の呼び出し慣例を使用することを Turbo Prolog に知らせています（3番目の例の DUBLIST.C と PLIST.PRO を見てください)。

値は参照引数として返されることに注意してください。フローパターンの詳細は，3番目の例題の複数フローパターンの指定についての説明を見てください。

例題1：2つの整数の和

最初の例題は，Turbo C の関数（2つの整数値の加算を行なうもの）と Turbo Prolog のモジュールを組み合わせて，C の関数の結果を現在のウィンドウに出力するものです。

Turbo C ソースファイル:CSUM.C

```
/*
   出力ルーチン zwf は，C の出力ルーチン printf に近い働きをします。
   これは，現在のウィンドウに出力を行ないます。
*/

extern void zwf(char *format. ...);

void sum_0(int parml,int parm2, int *res_p)
{
  zwf("This is the sum function: parml=%d, parm2=%d", parml, parm2);
  *res_p = parml + parm2;
}
/*  sum_0 の終わり  */
```

CSUM.C から CSUM.OBJ へのコンパイル

CSUM.C を作成し，セーブしたら，次にはコンパイルオプションを指定しなければなりません。Turbo C の場合，これには2通りの指定の方法があります。

1. TC（Turbo C 統合開発環境）のメニューから次のコンパイルオプションを選択します。

   O/**C**/**M**odel/**L**arge　(-ml)
   O/**C**/**O**ptimization/**J**ump optimization　(-o)
   O/**C**/**C**ode generation/**G**enerate underbars Off　(-u-)
   O/**C**/**O**ptimization/**U**se register variables Off　(-r-)

   以上のオプションを設定したら，メインメニューから Options/Save options を選択

して設定した内容をセーブし，Compile/Compile to OBJ を選択します。Turbo C は指定したオプションにしたがって CSUM.C をコンパイルし，CSUM.OBJ というオブジェクトモジュールを生成します。

2. Turbo C のメニューではなく，MS-DOS のコマンドラインでコンパイルしたいという方は，プロンプトから次のコマンドを入力してください。

```
tcc -ml -O -c -u- -r- csum
```

注意：Turbo Prolog はラージメモリモデルでのみコンパイルを行ないます。したがって Turbo C を Turbo Prolog にリンクする場合には，コンパイルオプション-ml（ラージメモリモデル）を使用しなければなりません。

Turbo Prolog ソースファイル:PROSUM.PRO

```
global predicates
  sum(integer,integer,integer) - (i,i,o) language c
          /* sum のフローパターンは (i,i,o) と宣言し,3番目の引数が戻り値,
             最初の2つは入力であることを指定する */

goal sum(7,6,X),write("Sum=",X).
```

PROSUM.PRO から PROSUM.OBJ へのコンパイル

PROSUM.PRO を作成してセーブしたら，オブジェクトファイル(.OBJ)にコンパイルして，Turbo C のオブジェクトモジュールとリンクしなければなりません。これには，まず Turbo Prolog のメインメニューで Complle を選択し，次に OBJ file を選びます。ソースファイルからオブジェクトファイルへのコンパイルが終わったら，後はリンクをして，この例題を実行してみるだけです。

CSUM.OBJ と PROSUM.OBJ とのリンク

Turbo Prolog と Turbo C のモジュールのリンクには，TC（Turbo C 統合開発環境），または TLINK（Turbo C のパッケージに含まれているスタンドアロンのリンカ）を使うことができます。tlink コマンドの後に，コマンドライン引数として，Turbo Prolog のメ

インモジュール，それ以外の関連モジュール，出力ファイル，ライブラリを指定します。特に指定しない限り，以上の引数は次の順序で並んでいなければなりません。

Turbo Prolog の初期化：

■ INIT.OBJ（Turbo Prolog 初期化モジュール）

Turbo Prolog メインモジュール：

■ゴールを含む Turbo Prolog のメインモジュール

関連モジュール：（このモジュールはどんな順序でもかまいません）

■アセンブラの.OBJ モジュール

■ Turbo C の.OBJ モジュール

■ Turbo Prolog の.OBJ モジュール

シンボルテーブルモジュール：

■ Turbo Prolog メインモジュールのシンボルテーブル名（これは，モジュールの最後に必らず置かなければなりません）

出力ファイル名：

■生成する実行可能ファイルのファイル名

ライブラリ：

■関連モジュールで必要とするルーチンが入っているライブラリをすべて指定します。ライブラリは次の順番で並べてください。

1. ユーザ定義ライブラリ
2. PROLOG.LIB
3. （必要であれば）EMU.LIB と MATHL.LIB
4. CL.LIB

この例では，Turbo Link（tlink コマンド）を使用して，以下の引数を指定しています。

■ Turbo Prolog のプログラム INIT.OBJ，PROSUM.OBJ

■ Turbo C のオブジェクトモジュール CSUM.OBJ

■ シンボルテーブル PROSUM.SYM，実行可能ファイル TEST.EXE

■ ライブラリ PROLOG.LIB，CL.LIB（浮動小数点数を使う場合は EMU.LIB と

MATHL.LIB も指定します)

注意：PROSUM.SYM はシンボルテーブルのファイルで，プログラム PROSUM.OBJ 内の変数名と変数の型が入っています。

最初の例題のリンクのコマンドは次の通りです。

```
tlink init prosum csum prosum.sym,test.exe,,prolog+cl
```

例題 2：数学ライブラリの使用

2番目は，最初の例題によく似たものです。ここでは，Turbo C の関数を2つ作成して，Turbo Prolog のプログラムと組み合わせる手順を説明しています。Turbo C の関数は，それぞれ別々のソースファイルとして作成します。CSUM1.C は2つの実数値を加算してその合計を返します。FACTRL.C は，整数の階乗を計算するプログラムです。Turbo Prolog のプログラム FACTSUM.PRO は，プログラムの結果を2つのウィンドウに出力します。この例では，Turbo C のラージメモリモデルの数学ライブラリ（MATHL.LIB）を使用します。

Turbo C ソースファイル：CSUM1.C

```
extern void zwf(char *format, ...);
void sum_0(double parm1, double parm2, double *res_p)
{
     *res_p = parm1+parm2;
     zwf("This is the sum function: parm1=%f, parm2=%f, result=%f",
                                   parm1, parm2, *res_p);
}
```

Turbo C ソースファイル：FACTRL.C

```
void factorial_0(int top, double *result)     /* 階乗の計算 */
{
   double x;
   int    i;

   if(top<1)
   {
      *result = 0.0;
      return;
   }
   *result = 1.0;
   x = 2.0;
   while (top-- > 1)
   {
      *result = *result * x;
          x++;
   }
}    /*  factorial_0 終わり  */
```

CSUM1.C, FACTRL.C から.OBJ へのコンパイル

最初の例題と同様，別のモジュールや Turbo Prolog のメインプログラムとリンクするには，まず Turbo C のモジュールをオブジェクトファイル（.OBJ）にコンパイルします。Turbo C のメニューでコンパイラオプションを設定して **O**ptions/**S**ave options でセーブし，C の各ソースファイルごとに Compile/Compile to OBJ コマンドを実行しても，tcc コマンドを使ってコマンドラインから2つのソースファイルをコンパイルしてもかまいません。どちらにしても，少なくとも以下のオプションを指定する必要があります。

O/**C**/**M**odel/**L**arge　(-ml)
O/**C**/**O**ptimization/**J**ump optimization　(-o)
O/**C**/Code generation/**G**enerate underbars Off　(-u-)
O/**C**/**O**ptimization/**U**se register variables Off　(-r-)

Turbo Prolog ソースファイル：FACTSUM.PRO

FACTSUM.PRO は Turbo Prolog のメインプログラムです。このプログラムはウィンドウを2つ生成します。1つは Turbo　C のモジュールからの出力を表示し，もう1つは Turbo Prolog のプログラムの出力を表示します。プログラムおよび各モジュールは次の順序で働きます。

1. FACTSUM.PRO は，プロンプトを出して整数 *Int* の入力を要求します。この値は Turbo Prolog から FACTRL.C に渡されます。
2. FACTRL.C の中の C の関数 *factorial* は，*Int* の階乗 *Result* を FACTSUM.PRO に返します。
3. FACTSUM.PRO は，*Result* をウィンドウに書き出し，もう一度プロンプトを出して数値の入力を求めます（今度は実数です）。
4. FACTSUM.PRO は，2度目に入力された数値 Real と **2.**で計算した階乗 *Result* を，モジュール CSUM1.C に渡します。
5. CSUM1.C 中の C の関数 *sum* は，*Real* と *Result* を加算して，その合計 *Sum* を FACTSUM.PRO に返します。
6. FACTSUM.PRO は *Sum* をウィンドウに書き出し，プログラムは終了します。

FACTSUM.PROは，次のようになります。

```
/*
   Turbo C モジュールの宣言は，Turbo Prolog モジュールや
   データベース（もしあれば）の宣言の後ろに置かなければならない。
   すべてのグローバルモジュールは，グローバルな述部 (predicate) として
   Turbo Prolog から呼ぶ必要があり，
   またフローパターンと言語指定をつけておかなければならない。
*/
global predicates
   sum(real,real,real) - (i,i,o) language c
   factorial(integer,real) - (i,o) language c
/*
   外部の clauses (Turbo C モジュール) を持つだけの簡単な例なので，
   goal セクションは必要ない。なお，実際のアプリケーションでは，
   これ以外に clauses セクションが必要。
*/
goal
   makewindow(1,49,31,
            " A Turbo Prolog window to the Turbo C program ",0,0,15,80),
   makewindow(2,47,3,
            " A Turbo Prolog window to the Turbo Prolog program ",15,0,10,80),

   /* 最初の入力のためのプロンプト */
   write("Enter an integer; Turbo C will calculate the factorial: "),
   readint(Int),nl,
   shiftwindow(1),     /* 出力ウィンドウをTurbo C用に変更 */

   /* Turbo C のモジュール factorial を呼び出して階乗を計算 */
   factorial(Int,Result),
   shiftwindow(2),     /* 出力ウィンドウをTurbo Prolog用に変更 */

   /* 2番目の入力のためのプロンプト */
   write("Enter a real number to add to the factorial: "),
   readreal(Real),nl,
   shiftwindow(1),     /* 出力ウィンドウをTurbo C用に変更 */

   /* Turbo Cのモジュール sum を呼び出して階乗を計算 */
   sum(Result,Real,Sum),
   shiftwindow(2),     /* 出力ウィンドウをTurbo Prolog用に変更 */

   /* 最初の計算結果をウィンドウに表示 */
   write("The factorial of ",Int," is ",Result),nl,

   /* 2番目の計算結果をウィンドウに表示 */
   write("The result: ",Result," + ",Real," = ",Sum),nl.
```

FACTSUM.PROからFACTSUM.OBJへのコンパイル

最初の例と同様に，他のモジュールとリンクする前に，Turbo Prologのソースファイル

をオブジェクトファイル（.OBJ）にコンパイルしなくてはなりません。前と同じように，Turbo Prolog のメインメニューで Compile/OBJ file を選んで，プログラムをコンパイルしてください。

CSUM1.OBJ, FACTRL.OBJ, FACTSUM.OBJ のリンク

リンクコマンドで指定する引数は以下のとおりです。

- ■ Turbo Prolog のオブジェクトモジュール INIT.OBJ，FACTSUM.OBJ
- ■ Turbo C のオブジェクトモジュール CSUM1.OBJ，FACTRL.OBJ
- ■出力ファイル FACTSUM.SYM（シンボルテーブル），SUM.EXE（実行可能ファイル）
- ■ライブラリ PROLOG.LIB，EMU.LIB，MATHL.LIB，CL.LIB

モジュールをリンクするコマンドは次のようになります。

```
tlink init factsum factrl csum1 factsum.sym,sum,,prolog+emu+mathl+cl
```

例題3：フローパターンとメモリ割り当て

次のプログラムは，Turbo Prolog の functor と list を Turbo C で作成し，この新たにつくった構造を Turbo　Prolog に返すコードを示しています。またこの例では，Turbo Prolog のグローバルスタックにメモリ領域を割り当てる方法についても説明しています。*list* は3つの要素からなる再帰的な構造です。また *functor* は2つのメンバからなる C の構造体です（この例題の後でもっと詳しく説明します）。

- Turbo C のモジュール DUBLIST.C は3つの関数からなっています。最初の2つの関数は，整数のリストを受けとって構造体を返すものと構造体を受けとって整数のリストを返すものです。3つ目の関数は整数 n を受けとって整数2つからなるリスト（1つ目が *n*，2つ目が2*n*）を返します。
- ここで重要なのは，各々の Turbo Prolog のグローバルな述語に対して複数のフローパターンを指定できるということです。この場合，各フローパターンに対して対応する Turbo C の関数が必要です。この例では，*clist_0* が最初のフローパターン（i,o）に，*clist_1* が2つ目のフローパターン（o,i）に，それぞれ対応していなければなりません。

```
global predicates
    clist(ilist,ifunc) - (i,o) (o,i) language c
```

- （i,o）は，*ilist* が Turbo C の関数 *clist_0* に渡されることと，*ifunc* は Turbo C の関数 *clist_0* 内で定義される構造体へのポインタであることを示しています。また，（o,i）は，*ifunc* が *clist_1* に渡されることと，*ilist* が *clist_1* 内で定義される構造体へのポインタであることを示します。
- Turbo Prolog のグローバルな **domain** でさらに別のフローパターンを指定する場合は，そのフローパターンを扱う関数 *clist_2* が必要となります。

Turbo C ソースファイル：DUBLIST.C

```
void fail_cc(void);
void *alloc_gstack(int size);

struct ilist {
   char functor;            /* リスト要素のタイプ */
                            /* 1 = リスト要素, 2 = リストの終わり */
```

```
    int  val;                  /* 実際の要素 */
    struct ilist *next;        /* 次のノードへのポインタ */
};

struct ifunc {
    char type;                 /* functor のタイプ */
    int  value;                /* functor の値 */
};

void clist_0(struct ilist *in, struct ifunc **out) {
    if (in->funtor !=1)  fail_cc();     /* リストが空であれば失敗 */
    *out = alloc_gstack(sizeof(struct ifunc));
    (*out)->value = int->val;           /* out を f(X) にセット */
    (*out)->type  = 1;                  /* functor のタイプをセット */
}

void clist_1(struct ilist **out, struct func *in) {
    int temp = 0;
    struct ilist *endlist = alloc_gstack(sizeof(struct ilist));
    endlist->functor = 2;
    temp = in->value;
    temp += temp;
    *out = (struct ilist *) palloc(sizeof(struct ilist));
    (*out)->val = temp;          /* リストとして [2*X] を返す */
    (*out)->functor = 1;         /* リストのタイプをセット．これを */
                                 /* 行なわないと意味のある値が返されない */
    (*out)->next = endlist;
}

void dublist_0(int n, struct ilist **out) {
    /* この関数はリスト [n,n+n] を作り出す */

    struct ilist *temp;
    struct ilist *endlist = alloc_gstack(sizeof(struct ilist));

    endlist->functor = 2;
    temp = alloc_gstack(sizeof(struct ilist));
    temp->val = n;               /* リストの最初の要素を n にセット */
    temp->functor = 1;
    *out = temp;

    /* 次にリストの2番目の要素を割り当てる */
    temp = alloc_gstack(sizeof(struct ilist));
    temp->val = n + n;           /* 2番目の要素に n+n を代入 */
    temp->functor = 1;
    temp->next = endlist;        /* 2番目の次のノードとして  */
                                 /*   リストの終わりをセット  */
    (*out)->next = temp;
}
```

Turbo C から Turbo Prolog を呼び出す

Turbo Prolog から Turbo C で書いた述語を呼び出せるだけでなく，Turbo C から Turbo Prolog の述語を呼び出すこともできます。グローバルな述語が Turbo Prolog の中で language c として宣言されており，その述語に対する Turbo Prolog の節があれば，Turbo Prolog は Turbo C から呼び出せるルーチンを生成します。

次の Turbo Prolog プログラムは，C 言語のグローバルな述語 ***message*** と ***hello _c*** を宣言しています。述語 ***message*** は，C ソースコードの中で関数名 *message _0* を使うことによって，C モジュールから呼び出すことができます。

```
global predicates
  message(strign) - (i) language c
  hello_c - language c

clauses
  message(S) :-
  makewindow(13,7,7,"",10,10,3,50),
  write(S), readchar(_),
  removewindow.

goal
  message("Hello from Turbo Prolog"),
  hello_c.
```

この例の **goal** セクションは Turbo C 関数 *hello _c* を呼び出しており，*hello _c* はメッセージを表示するために Turbo Prolog の述語 ***message _0*** を呼び出します。

```
void message_0(char * str);

void hello_c_0(void)
{
    message_0("Hello from Turbo C");
}
```

この機能を使えば，Turbo Prolog の強力なライブラリに他の言語から容易にアクセスすることができます。

Turbo Prolog モジュール内でユーザ独自のライブラリルーチンを簡単に定義することができます。

```
project "dummy"                    /* ここでは独自のプロジェクト名を使う */

global predicates
  myfail language c as "fail"
  mymakewindow(integer,integer,integer,string,integer,integer,integer,integer)
      - (i,i,i,i,i,i,i,i) language c as "makewindow"
  myshiftwindow(integer) - (i) language c as "shiftwindow"
  myremovewindow language c as "removewindow"
  write_integer(integer) - (i) language c as "write_integer"
  write_real(real) - (i) language c as "write_real"
  write_string(string) - (i) language c as "write_string"
  myreadchar(char) - (o) language c as "myreadchar"
  myreadstring(string) - (o) language c as "myreadstring"

  extprog language c as "extprog"

clauses
  myfail :- fail.
  mymakewindow(Wno, Wattr, Fattr, Text, Srow, Scol, Rows, Cols) :-
      makewindow(Wno, Wattr, Fattr, Text, Srow, Scol, Rows, Cols).
  myshiftwindow(WNO) :- shiftwindow(WNO).
  myremovewindow :- removewindow.
  write_integer(I) :- write(I).
  write_real(R) :- write(R).
  write_string(S) :- write(S).
  myreadchar(CH) :- readchar(CH).
  myreadline(S) :- readline(S).
```

次に示すＣ関数 *extprog* は，上の新しいライブラリルーチンをどうやって使うかを示しています。*extprog* は Turbo Prolog のウィンドウを作成し，そこでなんらかの読み書きを行ないます。

```
void makewindow(int who, int wattr, int fattr, char *title,
                int row, int srow, int scol);
void write_string(char *text);
void readchar(char *ch);
void readline(char *in_str[]);
void removewindow(void);

void extprog(void)
{
    char dummychar;
    char *Name;

    makewindow(1,7,7,"Hello there",5,5,15,60);
    write_string("¥n¥nIsn't it easy";
    readchar(&dummychar);
    write_string("¥nEnter your name: ");
    readline(&Name);
    write_string("¥nYour name is: ");
    write_string(Name);
    readchar(&dummychar);
    removewindow();
}
```

Turbo C から Turbo Prolog を呼び出す際の唯一の制限は，Turbo Prolog は自分でヒ

ープとスタックをセットアップする必要があるために，Turbo Prolog プログラムがメインモジュールでなければならないという点です。

list, functor

Turbo Prolog の list と functor は，DUBLIST.C を見てわかるように Turbo C モジュール内の構造体です。

list は，要素を3つ持つ再帰的な構造です。最初の要素はノードのタイプを示し，このデータ構造がリストの要素のときは1，リストの終わりを示すときは2になります。2番目の要素は実際の値です。この要素は Turbo Prolog の要素と同じ型でなければなりません。たとえば，実数のリストであれば次のようになります。

```
struct alist {
    char   funct;
    double elem;     /*  リストの要素は実数  */
    struct alist *next;
}
```

3番目の要素は，次のノードへのポインタです。

Turbo Prolog の functor は，2つのメンバからなる C の構造体です。最初のメンバは Turbo Prolog の **domain** 宣言に対応しています（Turbo Prolog の **domain** 宣言の詳細については，Turbo Prolog Reference Guide を参照してください）。たとえば，次のように宣言すると，

```
domains
    func = i(integer); s(string)
predicates
    call(func)
goal
    call(X)
```

func は Turbo Prolog の functor で，2つの型を持っています。1番目は整数，2番目は文字列です。したがってここでは，Turbo C の構造体のデータ型を示すメンバは1か2のどちらかの値をとります。1は1番目の型に，2は2番目の型に対応しています。

Turbo C の構造体の2番目の要素は，functor で対応する要素の実際の値です。これは引数の取り得る型からなる共用体として定義されています。

```
union val {
    int    ival;
    char   *svar;
};
struct func {
    char   type;          /* type は1か2の値をとり, */
                          /* Turbo Prolog の domain 宣言に対応する */
    union val value;      /* functor の要素の値 */
}
```

注意：メモリの取り扱いには，関数 **alloc _gstack**, **_malloc**, **_free** を使わなければなりません。次のような場合に，こうした関数が必要になります。

1. Turbo C の構造体を Turbo Prolog のスタックやヒープに置くためにメモリを割り当てる場合。
2. Turbo Prolog のヒープ上のメモリを解放する場合。

alloc _gstack を使った場合，割り当てたメモリ領域を越えてしまうような Turbo Prolog のバックトラックが起こると，メモリは自動的に解放されます。各関数の Turbo C での構文は次の通りです。

```
void *alloc_gstack(unsigned size);      /* スタックに記憶領域を確保 */
void *_malloc(unsigned size);           /* ヒープに記憶領域を確保 */
void _free(void *ptr, size);            /* ヒープ領域を解放 */
```

次は，Turbo Prolog のメインモジュール PLIST.PRO です。これは DUBLIST.C の中の関数を呼んで，その結果を出力します。

```
domains
  ilist = integer*
  ifunc = f(integer)

global predicates
  clist(ilist,ifunc) - (i,o) (o,i) language c
  dublist(integer,ilist) -(i,o) language c

goal
  clearwindow,
  clist([3],X),           /*   f(3) に X を結びつける */
  wtite("X = ",X),nl,
  clist(Y,X),             /*    [6] に Y を結びつける */
  wtite("Y = ",Y),nl,
  dublist(6,Z),           /* [6,12] に Z を結びつける */
  wtite(Z),nl.
```

DUBLIST.C のコンパイル

前の2つの例題と同様に，Turbo Prolog のメインモジュール PLIST.PRO とリンクする前に，Turbo C のモジュール DUBLIST.C をオブジェクトファイルにコンパイルしておきます。そのあと，次のようにしてリンクを行ないます。

```
tlink init plist dublist plist.sym,dublist,,prolog+emu+mathl+cl
```

例題 4：3D 棒グラフの作画

最後の例題では，C のモジュールと Prolog のモジュールとをコンパイル・リンクして，AI の柔軟性と C の持っているグラフィックス機能を組み合わせたプログラムを作ります。ソースコードには次のものが含まれています。

■別のファイルからの入力データを使って棒グラフを描く Turbo　C のモジュール CBAR.C
■ユーザにデータの入力を要求する Turbo Prolog のメインモジュール PBAR.PRO

Turbo C ソースファイル：CBAR.C

ここで使うソースコードは，マスターディスクの中に入っている CBAR.C です。

CBAR.C のコンパイル

これまでの例題と同様, Turbo Prolog のメインモジュール PBAR.PRO とリンクする前に，Turbo　C のモジュール CBAR.C をオブジェクトファイルにコンパイルしておきます。

Turbo Prolog のプログラム：PBAR.PRO

PBAR.PRO も，ソースコードはマスターディスクに入っています。PBAR はプロンプトを出して，棒グラフの作成・セーブ・ロード・作画を行なう Turbo Prolog のプログラムです。

棒グラフを作成したいときには，このプログラムはユーザの指定にしたがってウィンドウ内での各々のバーの位置を決め，バーを描くために Turbo C のモジュールを呼び出します。各バーを作画した後，データベースにアサーション(Prolog の用語で挿入のことです)されます。

棒グラフのセーブを選択することもできます。この場合，PBAR は後で使用できるように現在の棒グラフをファイルにセーブします。

ロードのオプションを選択すると，PBAR は現在の棒グラフを消去し，ファイルからユ

ーザの指定した棒グラフをロードします。

最後のオプションーdraw（作画）を選択すると，PBARはデータベースの内容にしたがってTurbo CのCBARモジュールを再帰的に呼び出し，現在データベースに指定してある通りに棒グラフを作画します（Turbo Cディスク中のPBAR.PRO参照）。

PBAR.PROからPBAR.OBJへのコンパイル

例題1と同様，Turbo Cのモジュールとリンクする前に，Turbo Prologのメインモジュールのソースファイルをオブジェクトファイルにコンパイルします。

PBAR.OBJとCBAR.OBJとのリンク

次に示すリンクコマンドで，PBAR.OBJとCBAR.OBJ（Turbo Cのモジュールもあらかじめコンパイルしておきます）とをリンクします。リンクコマンドでは以下のものを指定します。

- Turbo Prologのオブジェクトモジュール INIT.OBJ，PBAR.OBJ
- Turbo Cのオブジェクトモジュール CBAR.OBJ
- シンボルテーブル PBAR.SYM，出力ファイル BARCHART.EXE（実行可能ファイル）
- ライブラリ PROLOG.LIB，CL.LIB

リンクコマンドは次のようになります。

```
tlink init pbar cbar pbar.sym,barchart,,prolog+cl
```

最後に

以上4つの例で，Turbo Prolog のモジュールを Turbo C のプログラムとリンクする方法について説明してきました。Turbo Prolog のプログラミングが得意で，C のプログラミングについてもっと知りたいという方は，このマニュアルの第6章と第7章を読んでみてください。C は得意なので，Turbo Prolog についてもっと知りたいというのであれば，Using Turbo Prolog（P. Robinson 著，McGraw-Hill 社）などの Turbo Prolog の解説書を読んでみるとよいでしょう。

第11章

Turbo C 言語リファレンス

Cに関する参考文献というと，昔から Brian W. Kernighan と Dennis M.Ritchie による**「プログラミング言語 C(初版)」**（以降では，著者の頭文字をとって"K&R"と略記しています）ということになっています。この本はCに対する完全な規格は定義しておらず，その仕事は米国標準局（ANSI）がなすべきものとして残されてきました。K&R では最低限の規格が規定されています。したがって，K&R の中で示された機能だけを使っているプログラムは，K&R の定義をサポートする処理系ならどんなものを使ってもコンパイルできるはずです。

Turbo C は K&R の定義だけではなく，ANSI 標準化案の大部分もサポートしています。こうした中で，Turbo C は，古くからあるC言語に新しい機能を追加し，また柔軟性とパワーをより大きくすることによって，C言語を改良，拡張する道を探ってきたのです。ここでは，K&R の定義あるいは ANSI 標準化案をすべて記載する紙数はありませんので，K&R の定義に Turbo C が追加した部分についてだけ説明しています。なお，その追加が ANSI 標準化案に従ったものなのか，ボーランド社が独自に拡張したものであるのかについては示していません。

この章では……

この章では，K&R を参照しやすいように，なるべく K&R の付録 A**「C 参照マニュアル」**にそって説明しています。付録 A 全部を参照しているわけではありませんが，省略した節は，Turbo C と K&R の定義の間にほとんど差がないものと考えてください。また便宜上，ANSI 標準化案と Turbo C が K&R から拡張した部分については，K&R の説明順序ではなく，ANSI 標準化案の記述に従っている箇所もあります。

コメント（K&R 2.1）

K&Rの定義では，コメントはネスト（入れ子）することはできません。次のような例を見てみましょう。

```
/* myfunc() をコメントとしてくくりだす */

/*
   myfunc()
   {
       printf("This is my function¥n");      /* The only line */
   }
*/
```

ここでは，The only lineの直後でコメントは終わると解釈されます。したがって，対になっていない右カッコ }とコメントの終了記号*/はコンパイルエラーとなります。Turbo Cでも，デフォルトではコメントのネストはできませんが，コンパイラオプション-Cを使用すれば（または**O/C/Source**メニューの**Nested comments**を**On**にすれば），ネストになったコメントを含むプログラムをコンパイルすることができます。しかし，可搬性を保つためには，プログラムの一部分を#if 0と#endifで囲んでしまうのがよいでしょう。

Turbo Cでは，コメントは，マクロ展開の際に，空白文字一個に置き換えられます。処理系によっては，コメントが完全に取り除かれる場合もあり，トークンの結合に用いられることもあります。この章の「**トークンの置換**（**K&R 12.1**）」を参照してください。

識別名（K&R 2.2）

識別名というのは，変数，関数，データ型，その他ユーザが定義するものに付ける名前のことです。Cで識別名に使えるのは，英字（A～Z，a～z），数字（0～9），下線（ _ ）です。ただし，識別名の最初の文字は，英字または下線でなければなりません。

大文字と小文字は区別されます。つまり，識別名 *index* と *Index* は別個のものになります。Turbo Cでは，識別名の最初の32文字がプログラムによって認識されます。しかし，コンパイラオプション-i#を用いると，これを変更することができます。#は認識される文字数です。メニューオプションでは，**O/C/S/Identifier length**で指定します。

同様に，他のモジュールに由来するグローバルな識別名でも，有効なのは最初の32文字までです。グローバルな識別名の大文字と小文字の区別をするかどうかは，**Options/Linker**メニューの**Case-sensitive link**，またはTLINKの/cオプションで指定することがで

きます。ただし，pascal 型の識別名はどんな場合でも，リンク時には大文字と小文字の区別はされないということに注意してください。

キーワード（K&R 2.3）

表 11.1 は，Turbo C の予約語（キーワード）を示しています。予約語は識別名としては使用できません。先頭の AN という印は，ANSI 案が K&R に対して拡張した予約語を意味します。また TC は，Turbo C で拡張されたことを意味します。K&R では，entry と fortran という予約語について言及していますが，Turbo C では使用せず，また予約もされていません。

表 11.1 Turbo C で予約されているキーワード

TC asm	extern	return	TC _cs	TC _DH
auto	TC far	short	TC _ds	TC _DL
break	float	AN signed	TC _es	TC _DX
case	for	sizeof	TC _ss	TC _BP
TC cdecl	goto	static	TC _AH	TC _DI
char	TC huge	struct	TC _AL	TC _SI
AN const	if	switch	TC _AX	TC _SP
continue	int	typedef	TC _BH	
default	TC interrupt	union	TC _BL	
do	long	unsigned	TC _BX	
double	TC near	AN void	TC _CH	
else	TC pascal	AN volatile	TC _CL	
AN enum	register	while	TC _CX	

定数（K&R 2.4）

Turbo C では，K&R で定義されているすべての定数型を，いくらか強化してサポートしています。

整数型定数（K&R 2.4.1）

0～4294967295（10進数）の範囲の定数が使用可能です（負の定数は，符号なし定数に単項マイナス演算子がついたものと考えます）。8進表記と16進表記の定数も許されています。

定数の後ろに *L*（または *l*）という文字をつけると，その定数は long 型として表現され

ることになります。同様に，*U*（または *u*）という文字をつけると，その定数は unsigned 型になります。数値自体が65535より大きくなった場合は，（基数には関係なく）unsigned long になります。なお，1つの定数に，*L* と *U* 両方をつけてもかまいません。

表 11.2 に，3種類の表記法による定数の表現を示しました。

表 11.2　Turbo C の整定数

10 進定数

```
          0 - 32767          int
      32367 - 2147483647     long
 2147483648 - 4294967295     unsigned long
> 4294967295                 オーバーフローになるが警告は出力されない。
                             結果の定数は実際の値の低位部が保存される。
```

8 進定数

```
             00 - 077777          int
        0100000 - 0177777         unsigned int
       01000000 - 017777777777    long
  0100000000000 - 037777777777    unsigned long
> 037777777777                    オーバーフロー（同上）
```

16 進定数

```
     0x0000 - 0x7FFF          int
     0x8000 - 0xFFFF          unsigned int
    0x10000 - 0x7FFFFFFF      long
 0x80000000 - 0xFFFFFFFF      unsigned long
> 0xFFFFFFFF                  オーバーフロー（同上）
```

文字定数（K&R 2.4.3）

Turbo C では，2文字定数がサポートされています。たとえば，'An'，'¥n¥t'，あるいは'¥007¥007'などといったものです。これらの定数は16ビットの int 型の値として表現されます。最初の文字は低位バイトに，2番目の文字は高位バイトに入ります。このような定数は，他の C コンパイラでは使えない場合があるので注意してください。

'A'，'¥t'，あるいは'¥007'のような1文字定数も16ビットの int 型の値として表現されます。この場合，低位バイトの符号で高位バイトが埋められることになります。つまり，値が127（10進数）より大きい場合は，上位のバイトが-1［＝0xFF］になります。こうしたくなければ，デフォルトの char 型を符号なしであると宣言します（コンパイラオプション -K を使用するか，**O/C/C/Default** char type を Unsigned にする）。そうすれば，低位バ

イトの値に関係なく，高位バイトがゼロになります。

Turbo C は，ANSI 拡張の1つである文字コードの16進表記をサポートしています。たとえば，'¥x1F'や'¥X82'などと表記することができます。x と X はどちらでもよく，その後に1～3桁の数字を書くことができます。

Turbo C では，（これも ANSI 拡張の1つですが）エスケープシーケンスもサポートしています。エスケープシーケンスは，文字定数や文字列定数に挿入される値で，頭に円記号¥（IBM PC ではバックスラッシュ＼）がついたものです。**表 11.3** に，使用できるエスケープシーケンスの一覧を示してあります（＊がついているのは K&R に対する拡張です）。

表 11.3　Turbo C のエスケープシーケンス

シーケンス	値	文字	機能
＊¥a	0x07	BEL	ベル
¥b	0x08	BS	バックスペース
¥f	0x0C	FF	フォームフィード
¥n	0x0A	LF	ラインフィード
¥r	0x0D	CR	リターン
¥t	0x09	HT	タブ（水平）
＊¥v	0x0B	VT	垂直タブ
¥¥	0x5C	¥	円記号
¥'	0x2C	'	単引用符（アポストロフィ）
＊¥"	0x22	"	二重引用符
＊¥?	0x3F	?	疑問符
¥DDD		任意	DDD は1～3桁の8進数
＊¥xHHH	0xHHH	任意	HHH は1～3桁の16進数

＊ K&R に対する ANSI での拡張

注意：Turbo C では2文字定数が使用できます。したがって，3桁未満の8進数のエスケープシーケンスに続いて数字が1つあると，微妙な問題が生じてきます。このような場合，後に続く文字が8進数として許されるものであれば，Turbo C は，その文字はエスケープシーケンスの一部とみなします。たとえば，8や9は8進数の数字として許されないので，'¥258'という定数は，¥25と8からなる2文字定数と解釈されてしまいます。

浮動小数点定数（K&R 2.4.4）

K&R でも述べられているように，浮動小数点定数はすべて double 型であると定義されています。ただし，浮動小数点定数の末尾に *F* を添えれば，強制的に float 型とすることができます。

文字列（K&R 2.5）

K&R によると，文字列定数は厳密に1つの文字列単位，つまり2つのダブルクォートにはさまれたテキストです(例:"like this")。数行に渡って文字列定数を記述するには，継続文字として円記号（¥）を使用しなければなりません。

しかし Turbo C では，ひとつの文字列定数に対して複数の文字列単位を使用することができます。こうすると，文字列定数は文字列単位を連結したものになります。たとえば，次のようなことができます。

```
main()
{
    char     *p;
    p = "This is an example of how Turbo C"
        " will automatically¥ndo the concatenation for"
        " you on very long strings,¥nresulting in nicer"
        " looking programs.";
    printf(P);
}
```

このプログラムの出力は次のようになります。

```
This an example of Turbo C will automatically
do the concatenation for you on very long strings,
resulting in nicer looking programs.
```

ハードウェアの特性（K&R 2.6）

K&R によれば，基本的なデータ型のサイズと数値の範囲は，処理系に強く依存し，また多くの場合コンピュータのアーキテクチャによって決定されるものです。これは Turbo C にも，他の C コンパイラすべてに対しても当てはまります。**表**11.4は，Turbo C で使用されるデータ型のサイズと数値の範囲を示しています。

表 11.4　Turbo C のデータ型, サイズ, 範囲

型	サイズ（ビット）	範囲
unsigned char	8	0 - 255
char	8	-128 - 127
enum	16	-32768 - 32767
unsigned short	16	0 - 65535
short	16	-32768 - 32767
unsigned int	16	0 - 65535
int	16	-32768 - 32767
unsigned long	32	0 - 4294967295
long	32	-2147483648 - 2147483647
froat	32	3.4E-38 - 3.4E+38
double	64	1.7E-308 - 1.7E+308
long double	80	3.4E-4932 - 1.1E+4932
ポインタ	16	(near, _cs, _ds, _es, _ss ポインタ)
ポインタ	32	(far,huge ポインタ)

変換（K&R 6）

Turbo C は，あるデータ型を別の型に自動的に変換するための機構を備えています。この節では，K&R への追加項目と，処理系独自の部分について説明しています。

char,int,enum（K&R 6.1）

文字定数の整数型への代入は，16ビット全体の代入を行うことになります。これは，1文

字定数と2文字定数はどちらも16ビットの値として表現されているためです（「**文字定数（K&R 2.4.3）**」を参照してください）。デフォルトのchar型を（コンパイラオプション-Kを使って）符号なしとしていなければ，文字型（変数）を整数型に代入すると，自動的に符号の拡張が行なわれることになります。signed char型では，つねに符号の拡張がなされます。unsigned charがintに変換される場合，高位バイトは必らずゼロになります。

enum型の値は何の変更も受けずに，直接intに変換されます。同様に，intは直接enum型に変換されます。enum型の値と文字の間の変換は，intと文字の間の変換とまったく同じです。

ポインタ（K&R 6.4）

Turbo Cでは，プログラムの中で使われるポインタには，使用するメモリモデルや使用するポインタ型修飾子によって，いろいろな大きさのものが存在します。たとえば，あるメモリモデルでプログラムをコンパイルする際，ソースコード中のアドレシング修飾子（near，far，huge，_cs，_ds，_es，_ss）を使えば，そのメモリモデルによって決まるポインタの大きさを変更することができます。

ポインタは，ある特定の型を指すものとして宣言しなければなりません。たとえ，その型がvoidであってもそうです（voidは実際には，どんな型でも指すことができるポインタを意味しますが）。しかし，なんらかの型を指すものと宣言されていれば，ポインタは他の型を指すようにすることができます。Turbo Cでは，こうしたポインタの再割当てを行なうことができますが，ポインタがvoid以外の型として宣言されている場合には，コンパイラはポインタの再割当てが行なわれていると警告を出します。ただし，データ型を指すポインタを，関数を指すポインタに変換することはできません。この逆も同様です。

算術変換（K&R 6.6）

K&Rは"通常の算術変換"について述べており，算術式（**オペランド 演算子 オペランド**）の中に指定された値（オペランド）がどのように変換されるかのを，明確に示しています。算術式の中のオペランドの変換において，Turbo Cが使用している規則は次のようになります。

1. まず，3種類のchar型，short型，enum型，およびfloat型は，**表11.5**のように変換されます。この変換によって，演算子で結びつけられた2つの値は，int型（long

および unsigned 修飾子を含めて）か double 型のどちらかになります。

2. 一方のオペランドが long double であれば，もう一方も long double に変換されます。
3. 一方のオペランドが double であれば，もう一方も double に変換されます。
4. 2および3に該当しない場合，一方のオペランドが unsigned long であれば，もう一方も unsigned long に変換されます。
5. 4にも該当しない場合，一方のオペランドが long であれば，もう一方も long に変換されます。
6. 5にも該当しない場合，一方のオペランドが unsigned であれば，もう一方も unsigned に変換されます。
7. 2~6のどれにも該当しなければ，オペランドは両方とも int であるはずです。

式の結果の型は，2つのオペランドの型と同じ型になります。

表 11.5 通常の算術変換で使用される方法

型	変換後の型	方法
char	int	符号拡張
unsigned char	int	常に高位バイトをゼロで満たす
signed char	int	常に符号拡張
short	int	unsigned であれば unsigned int
enum	int	同じ値
float	double	仮数部を0で満たす

演算子（K&R 7.2）

K&R では単項プラス演算子（＋）はサポートされていませんが，Turbo C ではサポートされています。これは実際の効果は持っていませんが，単項マイナス演算子との対称性を提供しています。

通常 Turbo C は，整数型の式については，オペランドの交換可能な演算子（＊や二項演算子の＋）では入れ換えを行なって，なるべく効率的な式を得るために式を再構成します。ただし，浮動小数点型の式を再構成することはありません。したがって，浮動小数点式の評価順序を特定したいときには，カッコを使わなければなりません。

たとえば，*a*, *b*, *c*, *f* がすべて float 型の場合，次の式では，

```
f = a + (b + c);
```

必らず，まず式（*b* ＋ *c*）が評価され，その結果が *a* に加えられます。

型指定子と修飾子（K&R 8.2）

Turbo C では，K&R にはない次のような基本データ型がサポートされています。

- unsigned char
- unsigned short
- unsigned long
- long double
- enum（列挙型）
- void

Turbo C では int と short は等しく，ともに16ビットを占めます。詳しくは「**ハードウェアの特性（K&R 2.6）**」をご覧ください。

enum 型

Turbo C では，ANSI 標準で定義されている列挙型がサポートされています。

列挙データ型は，とびとびの整数値の組を記述するのに使用されます。たとえば次のように宣言することができます。

```
enum days { sun, mon, tues, wed, thur, fri, sat };
```

days の中に列挙されている名前は，整数の定数になります。最初の名前(*sun*)は自動的にゼロにセットされ，あとに続く名前には，それぞれ直前の名前に1を加えた値がセットされます (*mon*=1, *tues*=2,...)。名前に特定の値を指定することもできます。値の指定がない名前には，前に述べたように1ずつ加算した値が与えられます。宣言の例を次に示します。

```
enum coins { penny = 1, nickle = 5, dime = 10, quarter = 25 };
```

列挙型の変数には，int 型であればどんな値でも代入することができます。範囲を越えているかどうかのチェックはなされません。

void 型

K&R では，すべての関数は値を返すことになっており，返す値の型が宣言されていなければ，その関数は int 型であるとみなされます。Turbo C は，ANSI 案で定義されている void 型をサポートしています。これは，値を返さない関数であることを明示するために使用します。同様に，空の引数リストには，予約語である void を書くことができます。以下は，void 型の例です。

```
void putmsg(void)
{
    printf("Hello world¥n");
}

main()
{
    putmsg();
}
```

特別な構文として，関数が返す値を無視することを明示するために，式を void 型にキャストすることができます。たとえば，ユーザがキーを打つまで待ちたいけれども，何がタイプされたかは無視したい場合は，次のように書けばよいでしょう。

```
(void) getch();
```

また，void 型を指すポインタを宣言することができます。これは，何も指し示さないポ

インタを作り出すわけではなく，どんなデータ型でも指すことができるポインタを生成します。また，データの型は必らずしもわかっている必要はありません。どんなポインタも，キャストなしで void ポインタに代入することができます(逆も同様です)。ただし，void ポインタに対して間接演算子（*）を使用することはできません。なぜなら，基底型が定義されてない，つまりどんな型として値を取り出せばよいかがわからないからです。

signed 修飾子

K&R で定義されている型修飾子 long,short,unsigned に加えて，Turbo C ではさらに3つの修飾子をサポートしています。signed,const,volatile です（すべて ANSI 案で定義されています)。

signed 修飾子は unsigned の反対で，符号つき（2の補数）で格納される値であることを意味します。これは主として，プログラムを読みやすく，また完全にするために使われます。ただし，デフォルトの char 型を（signed ではなく）unsigned としてコンパイルする場合は，変数または関数を signed char 型に定義するためには，signed 修飾子を使用しなければなりません。signed のみを書くと，signed int を意味します。これは，unsigned が unsigned int を意味するのとちょうど同じです。

const 修飾子

const 修飾子は，ANSI 標準で定義されているように，そのオブジェクトに対する代入，あるいはインクリメントやデクリメントのような“副作用”を防止します。const ポインタを変更することはできませんが，それが指すオブジェクトは変更することができます。なお，const だけを書くと，const int とみなされます。次の例を見てください。

```
const float pi      = 3.1415926;
const       maxint  = 32767;
char *const str     = "Hello, world";    /* 定数ポインタ           */
char const *str2    = "Hello, world";    /* 定数文字列へのポインタ */
```

以上の宣言が行われていると，次の文はエラーになります。

```
pi  = 3.0;                      /* const への値の代入 */
i   = maxint--;                 /* const のデクリメント */
str = "Hi, there!";             /* str が他のものを指す */
```

ただし，strcpy(str,"Hi there!");という関数呼び出しは許されることに注意してくださ

い。これは，文字列リテラル"Hi,there!"から，strによって指されるメモリの位置へ1文字ずつコピーが行なわれるからです。

volatile 修飾子

volatile 修飾子もANSI標準に定義されているものですが，これはconstのほぼ反対です。これは，そのオブジェクトがプログラムによって変更を受けるだけでなく，プログラムの外側から，割り込みルーチンやI/Oポートによって変更を受ける可能性があることを意味します。オブジェクトがvolatileであると宣言することは，コンパイラに対して，そのオブジェクトの値はどんな時点でも変更される可能性があるので，それを含む式を評価する場合には，値に関する仮定をしないように警告することになります。それはまた，コンパイラがその変数をレジスタ変数にしないようにもします。

```
volatile int ticks;

interrupt timer()
{
    ticks++;
}

wait (int interval)
{
    ticks = 0;
    while (ticks < interval)
        ;                          /* 何もしない */
}
```

このルーチン（*timer*がハードウェアのクロック割り込みと結びつけられていることを仮定しています）を使うと，引数*interval*によって与えられた時間だけ待つことができます。高度な最適化を行なうコンパイラの場合は，whileループの中では*ticks*の値は変化しないはずなので，whileループの中では*ticks*の値をロードしない可能性があることに注意してください。

cdecl,pascal 修飾子

Turbo Cは，他の言語によって書かれたルーチンをプログラムが容易に呼び出せるようになっています。また，その逆も同様です。このように言語を結合（リンク）するときは，2つの重要な問題，つまり**識別名**と**引数の受け渡し**を扱わなければなりません。

Turbo C でプログラムをコンパイルするとき，プログラムの中のすべてのグローバルな識別名（関数名やグローバル変数名）は，リンクに備えてオブジェクトコードファイルの中にセーブされます。デフォルトでは，こうした識別名は，もとの文字ケースで(大文字・小文字がソースプログラムで記述されたとおりに）セーブされます。また，-u-オプション（**O/C/C/Generate underbars..Off**）を選択しなければ，下線記号（ _ ）が識別名の前に置かれます。

さらに，プログラム中で宣言されたすべての外部識別名（**extern**）は，同じ書式であるとみなされます。リンク時には（デフォルトでは）大文字/小文字は区別されます。したがって，異なったソースファイルで使用される識別名は，スペルも文字ケース（大文字か小文字か）も，どちらも正確に一致しなければなりません。

pascal

他の言語で書かれたコードを参照するような場合，状況によっては，このようなデフォルトの方法で名前をセーブしていると，問題が生じる可能性があります。

そこで Turbo C では，この問題を解決する手段を提供しています。どんな識別名も **pascal** 型として宣言することができます。**pascal** 型と宣言された識別名は，すべて大文字に変換され，また識別名の前に下線記号が付けられることもありません。(識別名が関数の場合には，引数の受け渡しの手順にも影響します。これについては，少し後の「**関数型修飾子（K&R 10.1.1）**」を参照してください)。**pascal** 型と宣言されていれば，ソースコードで使われる文字ケースは何であってもかまいません。リンク時に考慮されるのは大文字だけになります。

cdecl

-p オプション（**O/C/C/Calling convention..Pascal**）でコンパイルすることによって，ソースファイル中のすべてのグローバルな識別名を **pascal** 型にすることができます。しかしながら，ある識別名については文字ケースを保存し，下線記号を前に付けたままにしたい場合があります。特に，それが他のファイルからの C の識別名の場合です。

これを実現するには，その識別名を **cdecl** として宣言する必要があります（これも関数の引数の受け渡しに影響を及ぼします)。

たとえば，ヘッダファイル（stdio.h など）中のすべての関数は，**cdecl** 型として宣言されています。こうすることによって，たとえ-p オプションを用いてコンパイルしても，ライブラリルーチンとリンクできることを保証しています。

詳わしくは，この章の「**関数型修飾子（K&R 10.1.1）**」と第9章を参照してください。

near,far,huge 修飾子

Turbo Cには，間接演算子（*）に影響する（つまりデータを指すポインタを修飾する）3つの修飾子があります。near，far，hugeの3つです。この予約語の意味は，第12章で詳しく説明しているので，ここでは簡単に述べておきます。

Turbo Cでは，メモリモデルのうちの一つを使ってコンパイルを行ないます。使用するメモリモデルによって，データを指すポインタの内部書式が（他の事柄とともに）決定されます。スモールデータモデル（タイニィ，スモール，ミディアム）の場合には，すべてのデータポインタの長さは16ビットであり，またデータポインタはデータセグメント(DS)レジスタからのオフセットを与えます。ラージデータモデル（コンパクト，ラージ，ヒュージ）の場合は，データを指すすべてのポインタの長さは32ビットであり，ポインタはセグメントアドレスとオフセットの両方を与えます。

データモデルのサイズをどれか1つ決めて使うときに，デフォルトとは違うサイズまたは書式を持つポインタを宣言したい場合があります。このような場合に，修飾子near，far，およびhugeを使用します。

nearポインタの長さは16ビットしかありません。このポインタは，それ自身のセグメントアドレスとして，データセグメント（DS）レジスタの内容を使用します。これは，スモールデータモデルでのポインタのデフォルトです。nearポインタを使用すると，データはカレントの64Kバイトのデータセグメントに制限されます。

farポインタの長さは32ビットで，セグメントアドレスとオフセットの両方を含んでいます。これはラージデータモデルでのデフォルトです。farポインタを使用すれば，i8088/8086プロセッサの1Mバイトのアドレス空間のあらゆる場所のデータを参照することができるようになります。

hugeポインタの長さも32ビットで，やはりセグメントアドレスとオフセットの両方を含んでいます。farポインタと異なるのは，hugeポインタは必らず正規化されているということです。これに関する詳細は第12章で説明しているので，ここでは概要だけを述べます。

- hugeポインタに対しては，関係演算子(==，!=，<，>，<=，>=)はすべて，正確にまた期待どおりに機能します。farポインタに対してはうまく働きません。
- hugeポインタに対して算術演算を行なうと，正規化が行なわれるために，セグメントアドレスとオフセットの両方が影響を受けます。farポインタでは，影響を受けるのはオフセットだけです。
- hugeポインタ（が指しているアドレス）は，1Mバイトのアドレス空間全体を通して

増加させることができます。far ポインタ増加させていくと，結局のところは64K バイトのセグメントの出発点に戻ってしまいます。

■ huge ポインタに対して算術演算を行なうと，必らず正規化ルーチンが呼び出されるので，時間は余計にかかります。

構造体と共用体（K&R 8.5）

Turbo C は，構造体と共用体についても K&R に従っており，さらに次のような特徴が追加されています。

ワードアラインメント

-a コンパイラオプション（**O/C/C/A**lignment..Word）を使うと，Turbo C は構造体（あるいは共用体）に対して，ワードアラインメントに必要な‘パディング’（バイトの挿入）を行ないます。これによって，次の3つが保証されることになります。

■構造体はワード境界（偶数アドレス）から始まります。

■ char 型以外のメンバは，どれも構造体の始まりから偶数のオフセットに置かれます。

■構造体全体の長さが偶数バイトになるように，1バイトが（必要ならば）付け加えられます。

ビットフィールド

Turbo C では，ビットフィールドは signed int，unsigned int のどちらであってもかまいません。また，1～16ビットの領域を占めることができます。ビットフィールドは，ワード内の低位ビットから高位ビットへ割り当てられます。たとえば，次のビットフィールドは，次のページの図のように割り付けられます。

```
struct mystruct {
    int       i : 2;
    unsigned  j : 5;
    int         : 4;
    int       k : 1;
    unsigned  m : 4;
} a,b,c;
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
←	m		→	k	(未使用)				←	j			→	← i	→

整数フィールドは2の補数で表現されます。一番左のビットは符号ビットです。たとえば，（*a.k* のような）ビット長1の signed int のビットフィールドは-1と0の値しか持つことができません。ゼロでない値は，すべて-1と解釈されるからです。

文（K&R 9）

Turbo C では，K&R で定義されている文はすべて，変更なしでサポートされています。

外部関数定義（K&R 10.1）

Turbo C では，関数の中で行なわれた extern 宣言は当該ブロック内でのみ有効であり，宣言が行なわれたブロック範囲の外では認識されません。ただしコンパイラは，このような extern 宣言を記憶しておいて，以降で行なわれている同じオブジェクトに対する宣言と比較するために用います。

Turbo C は，ANSI が K&R の関数の定義に対して行なった拡張のほとんどをサポートしていますが，これには，関数プロトタイプの他に，関数型修飾子が含まれています。さらに Turbo C では，interrupt 型の関数のような，独自の強化も行なっています。

関数型修飾子（K&R 10.1.1）

extern や static に加えて，Turbo C では，関数の定義に関する特有な修飾子をいくつか用意しています。pascal，cdecl，interrupt，near，far，huge の6つです。

pascal 関数修飾子

pascal という修飾子は Turbo C 特有のもので，Pascal の引数の受け渡し方法を使用する関数（および関数を指すポインタ）のために用意されています。これを用いると，他の言語で書いたプログラムから呼び出せる関数を，Turbo C で書くことが可能になります。同様に，C のプログラムから，他の言語で書いた外部ルーチンが呼べるようになります。リンクに備えて，関数名はすべて大文字に変換されます。

注意：-p コンパイラオプション（**O/C/C/**Calling convention..Pascal）を使用すると，すべての関数（およびその関数を指すポインタ）は pascal 型として扱われます。また，pascal 型として宣言された関数は，pascal 型であることがわかっていれば，C のルーチンから呼ぶことができます。たとえば，次のように関数を宣言してコンパイルしたとすると，他の C プログラムはこれをリンクして呼び出すことができます。

```
pascal putnums(int i, int j, int k)
{
    printf("And the answers are:  %d, %d, and %d¥n",i,j,k);
}
```

ただしこの関数（*putnums*）を呼び出すプログラムでは，次のような宣言が行なわれていなければなりません。

```
pascal putnums(int i, int j, int k);

main()
{
    putnums(1,4,9);
}
```

pascal 型の関数では，**printf** などの関数とは違って，引数の個数を可変にすることはできません。したがって，pascal 型の関数の定義では，省略符号（...）は使用できません。（可変個の引数をとる関数の定義には省略符号を用いますが，これについては少し後の「関数プロトタイプ（**K&R 10.1.2**）」を参照してください。）

cdecl 関数修飾子

cdecl 修飾子も Turbo C に特有のものです。pascal 修飾子と同様に，これも関数や関数を指すポインタに対して使用されます。cdecl 関数修飾子の目的は，-p コンパイラオプシ

ョンを取り消して，その関数が通常のC関数として呼び出せるようにすることです。たとえば，-p オプションを設定してプログラムをコンパイルしたいけれども，**printf** を使いたい場合は，次のようにすればよいわけです。

```
extern cdecl printf();

putnums(int i, int j, int k);

cdecl main()
{
    putnums(1,4,9);
}

putnums(int i, int j, int k)
{
    printf("And the answers are:  %d, %d, and %d¥n",i,j,k);
}
```

プログラムを-p オプションでコンパイルする場合，実行時ライブラリから利用される関数は，すべて cdecl 宣言される必要があります。(stdio.h などの)ヘッダファイルの中を見ると，このような場合を想定して，どの関数もはっきりと cdecl 型として定義されているのがわかるでしょう。-p オプションを使う際には，**main** も cdecl として宣言しなければならないことに注意してください。これは，C スタートアップコードが **main** を呼び出すときには，必らずCの呼び出し方法を用いるからです。

interrupt 関数修飾子

Turbo C 特有のもう1つの修飾子は interrput です。interrput 型の関数は，8088/8086の割り込みベクタとともに使用されるものとして設計されています。Turbo C は，interrput 型の関数をコンパイルする際に，関数の入口と出口に特別なコードを付加して，AX,BX,CX,DX,SI,DI,ES,DS の各レジスタの値が保持されるようにします。他のレジスタ(BP,SP,SS,CS,IP) は，Cの呼び出しシーケンスの一部として，あるいは割り込み処理それ自身の一部として保存されます。一般的な interrput 定義は，次のようになります。

```
void interrupt myhandler()
{
    ...
}
```

このような割り込み関数は，void 型として宣言する必要があります。割り込み関数は，どんなメモリモデルの場合でも宣言することができです。ヒュージ以外のメモリモデルでは，DS がプログラムのデータセグメントに設定されます。ヒュージモデルでは，DS はモ

ジュールのデータセグメントに設定されます。

near,faar,huge 関数修飾子

near,far,huge 修飾子は Turbo C に特有のものです。これらは cdecl **あるいは** pascal と組み合わせることはできますが，interrupt と組み合わせることはできません。

interrupt 型でない関数は，near,far あるいは huge と宣言することができます。これは使用されるメモリモデルでのデフォルトの設定を無効にします。near 関数は near コールを使い，far および huge 関数は far コール命令を使います。

タイニィ，スモール，コンパクトメモリモデルでは，修飾子のない関数はデフォルトで near 型になります。ミディアムおよびラージメモリモデルでは，修飾子のない関数はデフォルトで far 型になります。ヒュージメモリモデルではデフォルトは huge 型になります。

huge 関数は far 関数とほとんど同じですが，DS レジスタの値が，その huge 関数が含まれているソースモジュールに対するデータセグメントのアドレスにセットされる点が異なります。

関数型 huge は，Turbo C と同じメモリ割り当てを使用しないアセンブリ言語のコードとのインターフェースを必要とするときに便利です。

関数プロトタイプ（K&R 10.1.2）

関数を宣言する際に，K&R が認めている関数宣言子は，関数名，関数型，一対の空のカッコから成るということだけです。引数が（もしあれば）宣言されるのは，関数そのものを実際に定義する時だけです。

ANSI 標準および Turbo C では，関数を宣言するために関数プロトタイプを使用することができます。これは，関数の引数に関する情報をもつ宣言子です。コンパイラはこの情報を利用して，関数の呼び出しが正しく行なわれているかどうかを調べます。コンパイラはさらに，この情報と，呼び出しの際の引数の型が一致していない場合に，引数を強制的に適切な型に変換します。たとえば，次のようなコードがあるとしましょう。

```
long  lmax(long v1, long v2);

main()
{
    int   limit = 32;
    char  ch    = 'A';
    long  mval;

    mval = lmax(limit,ch);
}
```

lmax に対する関数プロトタイプが与えられているので，このプログラムは代入に関する標準的な規則に従って，*limit* と *ch* を long 型に変換してから，*lmax* の呼び出しに備えてスタックに積みます。関数プロトタイプが宣言されていないと，*limit* と *ch* は，それぞれ整数と文字としてスタックに積まれます。この場合，*lmax* に渡されるスタックは，*lmax* が期待しているものと大きさと内容が一致しないので，問題が生じます。K&R では引数の型と個数のチェックがまったく行なわれませんが，関数プロトタイプを使うと，バグやその他のプログラミングエラーを見つけ出すことができるので，たいへん役に立ちます。

関数プロトタイプは，コードを読みやすくするのにも役に立ちます。たとえば，関数 **strcpy** は2つの引数，すなわちコピー元（*source*）とコピー先（*destination*）の2つの文字列をもっています。問題は，どっちがどっちかということです。関数プロトタイプを見れば，それがはっきりします。

```
char  *strcpy(char *dest, char *source);
```

ヘッダファイルが関数プロトタイプを含んでいる場合は，そのファイルをプリントアウトすれば，そうした関数を呼び出すプログラムを書くのに必要な情報の大部分を得ることができます。

カッコの中に，void しか入っていない関数宣言子は，関数に引数が1つもないことを意味します。

```
f(void);
```

引数が1つ以上ある場合には，カッコの中にはカンマで区切られた宣言子の並びが入ります。宣言子は，型だけを指定してもよく，識別名を入れてもかまいません。

```
func(int *, long);                /* 型のみ      */
func(int * count, long total);    /* 型と識別名 */
```

上の2つの宣言子では，関数 *func* は2つの引数を持つことになります。つまり，（*count* という名前の）int 型を指すポインタと，（*total* という名前の）long 型の整数です。識別名はあってもなくても，引数の型の不一致が起こったときに診断メッセージの中で使われる以外には，他の部分では影響はありません。

関数プロトタイプは通常，一定の個数の引数を受け入れるものとして関数を定義します。受け取る引数の個数が可変である C の関数（**printf** など）については，関数プロトタイプのカッコの中に省略記号（...）を指定することができます。たとえば次のようにします。

```
f(int *count, long total, ...);
```

この種のプロトタイプでは，固定の引数はコンパイル時に検査され，可変の引数はプロトタイプが存在しない場合と同じ手順で引き渡されます。

以下に，関数宣言子とプロトタイプの例をいくつか示します。

```
int  f();                /* intを返す関数.引数に関する情報はない.
                            K&Rの“古典的”スタイル. */

int  f(void);            /* intを返す関数.引数はない. */

int  p(int,long);        /* intを返す関数.
                            引数は2つ；1番目はint，2番目はlong. */

int  pascal q(void);     /* intを返すpascal型の関数.引数はない. */

char far *s(char *source, int kind);
                         /* charへのfarポインタを返す関数. 引数は2つで，
                            1番目はcharへのポインタ，2番目はint. */

int printf(char *format,...);
                         /* intを返す関数. 最初の引数はcharへの
                            ポインタで固定，2番目以降の引数は可変. */

int  (*fp)(int);         /* intを返す関数へのポインタ. intの引数1つ. */
```

次に，プロトタイプの指定がある場合とない場合で，Turbo Cが，関数呼び出しにおいて言語修飾子と仮引数をどう処理するかに関する規則を要約しておきます。

規則1：関数の定義における言語修飾子は，その関数のすべての呼び出しにおいて，関数の宣言で使用される修飾子に一致しなければならない。

規則2：関数の中で仮引数の値を変更することは許されているが，割り込み関数を除いて，この変更はその関数を呼んだ側のルーチンにおける実引数には何の影響も与えない（詳しいことは，第12章の「**割り込み関数**」を参照してください）。

関数プロトタイプがあらかじめ宣言されていない場合，Turbo Cは，「**算術変換（K&R 6.6）**」のところで述べた拡張規則に従って引数を変換してから，関数呼び出しに渡します。関数プロトタイプが宣言されている場合には，与えられた引数を宣言された引数の型に変換しますが，その変換は代入文と同じように行なわれます。

関数プロトタイプが省略記号(...)を含んでいる場合は，省略記号までは他のプロトタイプと同じように，与えられた関数の引数すべてを変換します。可変部分の引数が与えられている場合は，コンパイラはプロトタイプの指定がない場合に関する通常の規則に従って，関数の引数を拡張（変換）します。

プロトタイプを宣言している場合，（プロトタイプに省略記号を使っていないのであれば）引数の個数は一致しなければなりません。引数の型は，代入できる（通常の規則に従

って変換できる）ものでなければなりません。どんな場合でも，キャストを使えば，引数を関数プロトタイプが受け入れられる型に変換することができます。

こういった点を明らかにする例を以下に示します。

```
int   strcmp(char *s1, char *s2);     /* 完全なプロトタイプ */
char  *strcpy();                       /* プロトタイプなし   */
int   sampl(float, int, ...);          /* 完全なプロトタイプ */

samp2()
{
   char    *sx, *cp;
   double  z;
   long    a;
   float   q;

   if (strcmp(sx, cp))      /* #1: 正しい                             */
      strcpy(sx, cp, 44);   /* #2: Turbo C では許されるが可搬性は低い */
   sampl(3, a, q);          /* #3: 正しい                             */
   strcpy(cp);              /* #4: 実行時エラー                       */
   sampl(2);                /* #5: コンパイルエラー                   */
}
```

この例の5つの呼び出し（コメントの#1～#5）は，関数呼び出しとプロトタイプに関するの相違点を示しています。

#1では，**strcmp** の呼び出しは正確にプロトタイプと一致しており，まったく問題はありません。

#2では，**strcpy** の呼び出しには余分な引数があります（**strcpy** の引数は3個ではなく2個です）。この場合，Turbo C は少し時間を使って，余分な引数をプッシュするコードを作りだします。しかし，**strcpy** の引数に関する情報がコンパイラに与えられていないので，構文エラーは起きません。この呼び出しは可搬性が高いものではありません。

#3では，プロトタイプに従って，*sampl* の最初の引数は float 型に，2番目の引数は int 型に変換されます。この変換によって，かなりの有効数字が失なわれてしまう可能性があるので，コンパイラは警告を発するはずです。long 型を int 型に変換すると，上位ビットが切り落とされてしまうからです（int 型への明示的なキャストを使えば警告は出なくなります）。3番目の引数 *q* は，プロトタイプの省略記号に対応しているので，通常の算術変換に従って double 型に変換されます。**#3**は正しい呼び出しです。

#4では **strcpy** がもう一度呼び出されていますが，今度は引数の個数がたりません。これは実行時エラーを引き起こすことになり，プログラムを破壊するおそれがあります。コンパイラは（同じ関数の呼び出しで引数の数が前とは違っているのに!）何も言いません。これは，**strcpy** に対する関数プロトタイプが宣言されていないからです。

#5では *sampl* が呼び出されていますが，やはり引数の個数がたりません。*sampl* は少なくとも2つの引数を必要とするので，この文はコンパイルエラーになります。コンパイラ

は，呼び出しにおいて引数の個数が宣言よりも少ないというエラーメッセージを出します。

重要な注意：関数プロトタイプが実際の関数定義と一致していない場合，Turbo C は，その定義がプロトタイプと同一のファイルに存在する場合に限り，このエラーを検出します。プロトタイプのヘッダファイルを使ってさまざまなルーチンのライブラリを作るような場合，ライブラリをコンパイルする際には，そのヘッダファイルを必らずインクルードするのがよいでしょう。そうすれば，プロトタイプと実際の関数定義との間の矛盾を見つけることができます。

スコープ(通用範囲)に関する規則（K&R 11）

重複した識別名に関しては，Turbo C は K&R よりかなり寛容です。Turbo C には，4つの識別名のクラスが存在します。

変数名，typedef 名，列挙型のメンバ名は，それが定義されているブロックの中では，重複してはいけません。外部で宣言された識別名は，他の外部変数名と重複してはいけません。

構造体，共用体，列挙型のタグ名は，それが定義されているブロックの中では，重複してはいけません。関数の外部で宣言されたタグ名は，外部で定義された他のすべてのタグ名と重複してはいけません。

構造体，共用体のメンバ名は，その構造体または共用体の中の他のメンバ名と重複してはいけません。別々の構造体のメンバ名に関しては制限は何もありません。

goto ラベルは，それが宣言されている関数の中では重複してはいけません。

コンパイラ制御行（K&R 12）

Turbo C では，K&R に見られる制御コマンドをすべてサポートしています。これらはプリプロセッサ指令と呼ばれ，実際には#で始まるソース行です。#の前または後ろにはホワイトスペース（空白，タブ，改行）を置くことができます。

トークン（綴り）の置換（K&R 12.1）

Turbo C では，K&R が定義している# define と# undef をサポートしていますが，さら

に以下のような点が追加されています。

■次の識別名は，# define 指令と# undef 指令の中に現われてはなりません。

```
__STDC__
__FILE__
__LINE__
__DATE__
__TIME__
```

■マクロ定義内で##で区切られている2つのトークンは，(##の両側にホワイトスペースがあっても)結合されます。プリプロセッサはホワイトスペースと##を取り除き，2つのトークンを結合します。これは識別名を作り出すために使用されます。たとえば次の定義があると，

```
#define VAR(i,j)  (i ## j)
```

VAR(x,6)は(x6)に展開されます。これは(可搬性は低いけれども)ときどき用いられる(i/**/j)を使用する方法の代用になります。

■マクロ定義文字列内に置かれているネストしたマクロが展開されるのは，もとのマクロが展開されるときであって，そのマクロが定義されるときではありません。これは，マクロのネストと# undef の間の関連に影響を与えます。

■シンボル#は，マクロの引数の前に置くことによって，その引数を文字列化(文字列に変換)することができます。マクロが展開されるときに，**#仮引数**が**"実引数"**に置き換えられます。したがって，次のマクロ定義が与えられると，

```
#define TRACE(flag) printf(#flag "=%d¥n", flag)
```

次の文は，

```
highval = 1024;
TRACE(highval);
```

以下のようになり，

```
highval = 1024;
printf("highval" "=%d¥n", highval);
```

さらに，以下のようになります。

```
highval = 1024;
printf("highval=%d¥n", highval);
```

■他の処理系とは異なり，Turbo C は文字列および文字定数内のマクロ引数を展開しません。

ファイルのインクルード（K&R 12.2)

Turbo C では，K&R の# include 指令をサポートしていますが，さらに次のような特徴も備えています。# include "filename"という形を用いた場合に，プリプロセッサは，デフォルトディレクトリの中でインクルードファイルが見つからなければ，次にコンパイラオプション -I で指定されたディレクトリを検索します。# include <filename>という形を用いている場合は，-I で指定されたディレクトリだけが探索されます（メニューオプション **O/D**/Include directories に列挙してあるディレクトリは，コマンドラインオプション -Ipathname で指定されたディレクトリと同じです)。

ディレクトリの区切り記号（¥）を含む# include 文を表現する場合，マクロ展開を用いることができます。# include の後ろに識別名がある場合，プリプロセッサは定義されているマクロの中からそのテキストを検索します。しかし，# include の後ろの文字列が引用符（""）か不等号記号（<>）で囲まれている場合は，その中にマクロと同じテキストがあっても展開を行ないません。したがって次の場合では，

```
#define    myinclude    "c:¥tc¥include¥mystuff.h"
#include   myinclude
#include  "myinclud.h"
```

最初の# include 文は C:¥TC¥INCLUDE¥MYSTUFF.H を探し，2番目はデフォルトディレクトリの中から MYINCLUD.H を探します。

また，# include 文で使用されるマクロ内では，文字列リテラルの連結や，トークンの結合を使ってはなりません。マクロ展開によって，通常の# include 指令のように読めるテキストが生成されなければなりません。

条件つきコンパイル（K&R 12.3)

K&R の定義では，指定された数行をホワイトスペース1行に置き換えることによって条件つきコンパイルが実行されますが，Turbo C でもこれをサポートしています。こうして無視される行は，# if，# ifdef，# ifndef，# else，# elif，# endif 指令で始まる行，および以

上の指令の結果コンパイルされなくなる行です。条件つきコンパイル指令は，それが始まったソースファイルまたはインクルードファイルの中で必らず終わらなければなりません。

Turbo C は ANSI の演算子 defined（symbol）をサポートしています。この symbol が（# define によって）すでに定義され，そのあと（# undef によって）定義を解除されていなければ，この値は1（真）になります。そうでなければ，その値は0（偽）です。したがって，次の2つの指令は同じ意味を持ちます。

```
#if defined(mysym)
#ifdef mysym
```

defined は，# if 指令に続く複雑な式の中で繰り返し使用できるので，# ifdef 文を連ねる必要がなくなります。たとえば次のように書くことができます。

```
#if  defined(mysym) || defined(yoursym)
```

最後に，Turbo C では（ANSI とは違って）プリプロセッサ式の中で sizeof 演算子を使用することができます。したがって次のような記述が可能です。

```
#if (sizeof(void *) == 2)
  #define SDATA
#else
  #define LDATA
#endif
```

行制御（K&R 12.4）

Turbo C は，K&R が定義している# line 指令をサポートしています。マクロは# include 指令の中で展開されますが，同じように# line の中でも展開されます。

error 指令（ANSI C 3.8.5）

Turbo C は# error 指令をサポートしています。この指令は（はっきりと定義されてはいませんが）ANSI 案で示されています。書式は次のようになります。

```
#error errmsg
```

また，メッセージは下の形式で出力されます。

```
Fatal: filename line# Error directive: errmsg
```

一般的にこの指令は，コンパイル時の望ましくない条件をとらえるために，プリプロセッサの条件判断指令とともに指定します。通常（望ましい条件）の場合，条件判定は真にはならないようにしておき，条件が真となった場合には，コンパイラはメッセージを表示して処理を中断します。つまり，望みどおりでない場合に真となる条件判断部の中に#error 指令を入れておくわけです。

たとえば，*MYVAL* が定義されており(#define *MYVAL*)，その値は0か1のどちらかでなければならないとします。この場合，ソースプログラムの中に以下のような条件判断部を置いて，*MYVAL* の値が正しいものかどうかをテストすることができます。

```
#if (MYVAL != 0 && MYVAL != 1)
#error MYVAL must be defined to either 0 or 1
#endif
```

プリプロセッサは，#error 指令行の中のコメントを取り除き，残りのテキストを表示します。

pragma 指令（ANSI C 3.8.6）

Turbo C は# pragma 指令をサポートしています。この指令は (#error と同じで) ANSI 案ではあいまいにしか定義されていません。この目的は，次の形の“処理系独自の”指令を許すことです。

```
#pragma  指令名
```

pragma を用いることによって，その処理系が必要とする指令を何でも定義することができますが，# pragma をサポートする他のコンパイラに影響を与えることはありません。これは，そのコンパイラが指令名を認識できない場合は，その# pragma 指令を無視することになっているからです。

pragma inline

Turbo C は3つの# pragma 指令を認識します。その1つは，

```
#pragma inline
```

です。この指令は-B コンパイラオプションと同じです。これは，プログラム内にインラインアセンブリ言語コードが存在するということをコンパイラに知らせます(第9章を参照して下さい)。これはファイルの先頭に置くのが最良です。# pragma inline に出会うと，コ

ンパイラは自分で-Bオプションを指定して，（TASMを呼び出すために）コンパイル処理をリスタートする（最初からやり直す）からです。実際には，-Bオプションと#pragma inline指令の両方とも指定しないでおいてもかまいません。こうすると，asm文に出会ったときにコンパイル処理をリスタートします。このオプションと指令の目的は，コンパイル時間を節約することにあります。

#pragma warn

2番目の#pragma指令は

```
#pragma warn
```

です。この指令によって，明示的な-w*xxx*コンパイラオプション（あるいは**O/C/E/D**isplay warnings..On）を無効にすることができます。

たとえば，ソースコードに次のような指令が置かれていると，

```
#pragma warn +xxx
#pragma warn -yyy
#pragma warn .zzz
```

*xxx*警告は（**O/C/E/D**isplay warnings..Offとなっていても）出力され，*yyy*警告は出力されません。また，*zzz*警告はファイルのコンパイルが始まったときの値に戻されます。

この指令で使用する3文字の略語とそれに対応する警告は，リファレンスガイドの付録Cに一覧表があります。

#pragma saveregs

このpragmaは，huge関数に入ったときに，そのhuge関数がどのレジスタの値も変更しないことを保証します。この指令は，アセンブリ言語コードとのインターフェースをとるときに必要になることがあります。#pragma saveregsは，該当する関数定義の直前に置かなければなりません。これはその関数に対してのみ適用されます。

ヌル指令（ANSI C 3.7）

完全を期すために，ANSI案とTurbo Cはヌル指令を認識します。ヌル指令は，文字#のみを含む行から成り立つというだけです。この指令は常に無視されます。

標準マクロ名（ANSI C 3.8.8）

ANSI では，5つの標準マクロが処理系で利用できるように要求しています。Turbo C はこの5つ全部をサポートしています。各マクロ名の始めと終わりには，それぞれ下線が2つ（__）ついていることに注意してください。

__LINE__ 現在処理中のソースファイルの行番号を示す10進の定数。ソースファイルの最初の行は1と定義されています。

コンパイラが# include 指令や# line 指令を処理するときには，このマクロは変化します。

__FILE__ 現在処理中のソースファイルの名前を示す文字列リテラル。

コンパイラが# include 指令を処理するときには，このマクロは変化します。

__DATE__ プリプロセッサが現在のソースファイルの処理を開始した時の日付を示す文字列リテラル。

1つのファイルに__DATE__を複数入れた場合，処理にどんなに時間がかかっても，同じ値をもつことが保証されています。日付は *mmm dd yyyy* という書式で表現され，*mmm* は月（Jan，Feb，など），*dd* は日（1～31，1桁なら10の位は空白），*yyyy* は年（1987，1988，など）を表わします。

__TIME__ プリプロセッサが現在のソースファイルの処理を開始した時刻を示す文字列リテラル。

1つのファイルに__TIME__を複数入れた場合，処理にどんなに時間がかかっても，同じ値をもつことが保証されています。時刻は *hh*：*mm*：*ss* という書式をとり，*hh* は時（00～23），*mm* は分（00～59），*ss* は秒（00～59）を表わします。

__STDC__ ANSI 互換性（-A）オプション（**O/C/S/ANSI** keywords only..On）を指定してコンパイルしている場合は定数の1，そうでなければこのマクロは定義されません。

Turbo C 標準マクロ

Turbo C のプリプロセッサでは，使用可能なマクロがさらにいくつか定義されています。ANSI が規定しているマクロと同様に，各マクロの始まりと終わりには下線記号が2つ

(__) 付きます。

__TURBOC__	現在のTurbo Cのバージョン番号を示す16進定数。バージョン1.0は0x0100，バージョン1.5は0x0105のようになります。
__PASCAL__	-pオプションのシグナルです。-pオプションが使用されている場合は整定数1，そうでなければ定義されません。
__MSDOS__	どんな場合にも整定数1。
__CDECL__	-pオプションは使用されていない（**O/C/C/**Calling convention..C）ことを知らせます。-pフラグが使用されていなければ整定数1，そうでなければ定義されません。

次の6つのシンボルは，コンパイル時に選択されたメモリモデルを示すように定義されます。どんな場合でも1つだけが定義され，その他は定義されません。たとえば，スモールモデルでコンパイルする場合は__SMALL__だけが定義され，他は定義されません。したがって，#if defined(__SMALL__)は真に，#if defined(__HUGE__)（およびこの他）は偽になります。定義されるマクロの値は1です。

__TINY__	タイニィモデル
__SMALL__	スモールモデル
__MEDIUM__	ミディアムモデル
__COMPACT__	コンパクトモデル
__LARGE__	ラージモデル
__HUGE__	ヒュージモデル

アナクロニズム——時代遅れ（K&R 17）

K&Rで指摘されているCの時代遅れな点は，Turbo Cでは採り入れていません。

第12章

Turbo Cでの上級プログラミング

遅かれ速かれ，この章まで来られるとは思っていました。きっと，学校の勉強とは較べものにならない速さで，水を吸うスポンジのごとく知識を吸いあげながら，これまでの章を突破してきたに違いありません。そして今，ニューフロンティアを求めて，より深遠な領域へと探査の足を踏み出そうとお考えなわけです。心より歓迎申し上げます。

この章では……

ここでは，4つの大きな話題を採り上げています。

■まず，メモリモデルについてタイニィモデルからヒュージモデルまでお話しします。メモリモデルとはどんなものか，どうやって1つのメモリモデルを選択するか，なぜそのメモリモデルを使う（あるいは使わない）と決めるのか，という点を説明していきます。

■次は，他の言語と組み合わせてプログラムを開発するという問題です。Turbo C と Turbo Prolog とを組み合わせる場合については，すでに第10章で見てきました。ここでは，Pascal およびアセンブリ言語とのインターフェースについて説明します。

■そのあと，Turbo C での低レベルのプログラミングを3つの面から見ていきます。アセンブリコードのインライン展開，擬似変数，割り込み処理の3つについて述べます。

■最後に，浮動小数点数に関する問題を採り上げます。

それでは，さっそく始めましょう。

メモリモデル

メモリモデルとは何でしょうか。また，なぜそんなことを気にしなければならないのでしょうか。この質問に答えるには，今使っているコンピュータシステムについて少し知っておく必要があります。システムの中央処理装置（CPU）は，インテルのiAPX86ファミリィに属しているマイクロプロセッサです。おそらく，8088,8086,80186,80286,80386のいずれか（あるいはV30）でしょう。以降では，8086ということで話を進めていくことにします。

8086 のレジスタ

図 12.1　8086 のレジスタ

汎用レジスタ：	**AX**	**AH**	**AL**	アキュムレータ
	BX	**BH**	**BL**	ベース（インデックス）
	CX	**CH**	**CL**	カウント（ループ，その他）
	DX	**DH**	**DL**	データ（データの保持）
セグメントレジスタ：	**CS**			コードセグメントポインタ
	DS			データセグメントポインタ
	SS			スタックセグメントポインタ
	ES			エキストラセグメントポインタ
特殊用途レジスタ：	**SP**			スタックポインタ
	BP			ベースポインタ
	SI			ソースインデックス
	DI			デスティネーションインデックス

この図は，8086のレジスタの構成で，各レジスタの用途を簡単に示しています。この他に2つのレジスタ，IP（命令ポインタ）とフラグレジスタがありますが，Turbo C ではアクセスできないので省略してあります。

汎用レジスタ

汎用レジスタは，データの記憶や操作のために，最も頻繁に使用されるレジスタです。各レジスタは，それぞれ独自の機能を持っています。たとえば次のようなものがあります。

- ■多くの数学的演算は，AX を使わなければ行なえない。
- ■ BX は，far ポインタのオフセットを保持するのに使用される。
- ■ CX は，8086の LOOP 命令で使用される。
- ■ DX は，データを保持するために使用される。

しかし，4つのレジスタのどれを使っても行なえる操作は数多くあり，多くの場合レジスタを自由に選んで使用することができます。

セグメントレジスタ

セグメントレジスタは，4つのセグメントの開始アドレスを保持しています。次の節で詳しく説明しますが，セグメントレジスタの16ビット値を4ビット左にシフト（16倍）すれば，そのセグメントの実際の20ビットアドレスが得られます。

特殊用途レジスタ

8086は，特殊な用途のためのレジスタをいくつか持っています。

- ■ SI と DI は，汎用レジスタの機能の多くを持つほか，インデックスレジスタとして使用されます。Turbo C では，レジスタ変数を置く場所として使用されます。
- ■ SP は，現在のスタックの先頭を，スタックセグメントからのオフセット値で表わします。
- ■ BP は，第2のスタックポインタで，通常は引数を取り出すために，スタックのインデックスとして使用されます。

ベースポインタ（BP）は，C の関数の中で，引数と自動変数のベースアドレスとして使用されています。引数は BP からの正のオフセット値を持ち，その値は，メモリモデルと，関数の入口で退避されたレジスタの個数によって変化します。BP は常に，直前に退避された BP の値を指しています。引数を持たない関数や，引数の宣言がない関数は，BP を使っ

たり，退避したりすることはありません。

自動変数にはBPからの負のオフセット値が与えられ，1番目の自動変数が最も小さい(絶対値が最大の）負のオフセット値を持つことになります。

メモリセグメンテーション

8086マイクロプロセッサは，**セグメント化されたメモリアーキテクチャ**を持っています。そのアドレス空間は，全体としては1Mbありますが，1度には64Kしかアクセスできないようになっています。その64Kの連続したメモリのことを**セグメント**と呼んでいます。

では，いったい何個のセグメントが存在するのでしょうか。そして，それはどこに配置され，8086はどのようにしてその場所を把握するのでしょうか。

■8086には，4つの異なるセグメント（**コード**，**データ**，**スタック**，**エキストラ**）があります。コードセグメントには機械語命令が格納され，データセグメントにはデータが，スタックセグメントにはもちろんスタック情報がおさめられます。エキストラセグメントは，通常データセグメントの予備として使用されます。

■8086は，CS, DS, SS, ESという4つの16ビットセグメントレジスタを持っており，それぞれコード，データ，スタック，エキストラの各セグメントを指しています。

■セグメントはメモリ上のどの場所にも置くことができますが，そのスタートアドレスは，16（10進）の整数倍でなければなりません。この理由は，先に進むにつれて明らかになります。

アドレスの計算

それでは，8086がこれらのセグメントレジスタを使ってどのようにアドレスを計算するのかを説明しましょう。8086上の完全なアドレスは，セグメントアドレスとオフセットという2つの16ビット値から構成されています。たとえば，データセグメントのアドレス(DSレジスタの値）が2F84（16進）だとすると，データセグメントの先頭からのオフセットが0532（16進）であるデータの本当のアドレスを計算してみましょう。

まず，セグメントレジスタの値を4ビット（16進では1桁）左にシフトし，これにオフセット値を加えます。

こうしてできた20ビットの値が実際のアドレスになります。具体的には次のようになります。

```
DSレジスタ(シフト後): 0010 1111 1000 0100 0000  =  2F840
オフセット          :      0000 0101 0011 0010  =  00532

アドレス            : 0010 1111 1101 0111 0010  =  2FD72
```

セグメントの開始アドレスは常に20ビットの数値ですが，セグメントレジスタはそのうちの上位16ビットだけを記憶し，残りの4ビットは常に0であるとみなします。そのためセグメントは，必らずメモリ上の16バイトごとのアドレス(最後の4ビットがゼロ，つまり16進の最後の桁がゼロ）のところから始まることになります。

したがって，DS レジスタが2F84という値を持つ場合は，データセグメントは2F840というアドレスから始まります。ところで，16バイトが集まったものを**パラグラフ**と呼ぶことがありますが，この言葉を使えば，セグメントは必らずパラグラフの境界から開始するといえます。

アドレスは普通，**セグメント**：**オフセット**という形式で記述します。上の例であれば，2F84：0532と表記するわけです。なお，オフセットはオーバーラップする（重なりあう）ことができるので，あるアドレスを表わすセグメント：オフセットという書き方は一通りではないということに注意してください。たとえば，以下のアドレス表記はすべて同じメモリ位置を表わしています。

```
0000:0123
0002:0103
0008:00A3
0010:0023
0012:0003
```

セグメントはオーバーラップすることができます（そうしなくてもいいのですが)。4つのセグメントがすべて同じアドレスから始まっていてもかまいません。こうすると，プログラム全体の大きさが64K を越えることはありません。言いかえると，コード，データ，およびスタックのための領域は全部で64K だけということになります。

near,far,huge ポインタ

ポインタは，Turbo C の各メモリモデルにおいて，何をしなければならないのでしょうか。たくさんあります。選択したメモリモデルの種類によって，コードとデータに対して使用されるポインタのデフォルトの型が決められます。しかし，どんなメモリモデルの場合でも，ポインタ（あるいは関数）を特定の型として明示的に宣言することができます。**near**（16ビット)，**far**（32ビット)，**huge**（32ビット）の3つの型です。それでは，この3つの型を順に見ていきましょう。

near ポインタ

near (16ビット) ポインタは，アドレスの計算に際してセグメントレジスタの1つを利用します。たとえば，near ポインタが指している関数のアドレスは，そのポインタの16ビット値を，コードセグメント（CS）レジスタの内容を左にシフトしたものに加えることで得られます。同様に，near データポインタは，データセグメント (DS) レジスタに対するオフセットを持っています。near ポインタは，どんな算術計算の場合でもセグメントについては考える必要がないので，容易に操作することができます。

far ポインタ

far ポインタ (32ビット) は，セグメント内でのオフセットだけでなく，（もう1つの16ビット値）セグメントアドレスを持っており，これを左シフトしてオフセットに加えてできたアドレスがポインタの指す場所になります。far ポインタを使うと，複数のコードセグメントを操作することができるので，64K バイトを越えるプログラムを作ることができます。また，far ポインタで64K バイト以上のデータにアクセスできるようになります。

ただし，データに対して far ポインタを使う場合，ポインタ操作については格別な注意を払う必要があります。「**アドレスの計算**」で述べたように，同じアドレスを表わすセグメント：オフセットの組合せはたくさんあります。たとえば，0000：0120，0010：0020，0012：0000 という far ポインタは，すべて同じ20ビットアドレスを表わしています。3つの far ポインタ変数 (*a*,*b*,*c*) が，それぞれこの3つの値を持つとすると，以下の式はすべて偽になります。

```
if (a == b) .....
if (b == c) .....
if (a == c) .....
```

同様な問題は，far ポインタの比較に，>，>=，<，<=などの演算子を使う場合にも生じます。この場合，オフセットだけが (unsigned として) 比較されるので，a,b,c の値が前と同じだとすると，以下の式はすべて真になります。

```
if (a > b) .....
if (b > c) .....
if (a > c) .....
```

==（等しい）と!=（等しくない）の場合は，32ビット値を（完全なメモリアドレスではなく）unsigned long として比較を行ないます。一方，大小を比較する演算子（<=，>=，<，>）の場合は，オフセット値だけを比較します。

==と!=は，ヌル（NULL）ポインタ（0000：0000）との比較のために，32ビット全部を必要とします。等しいかどうかの判定にオフセット値だけを使ったのでは，プログラマの意図に反して，オフセットが0000のポインタはすべてヌルポインタと等しくなってしまうからです。

注意すべき点が，もう1つあります。far ポインタにある値を加えると，変化するのはオフセットだけです。オフセットが FFFF（16ビットの最大値）を越えるような値を加えると，ポインタは元のセグメントの最初に戻ってしまいます。たとえば，5031：FFFF に1を加えると，（6031：0000ではなく）5031：0000になります。同様に，5031：0000から1を引くと，（5030：000F ではなく）5031：FFFF になります。

ポインタ同士の比較を行なう場合に一番安全なのは，（セグメントアドレスが共通な）near ポインタか，次に説明する huge ポインタです。

huge ポインタ

huge ポインタも32ビットで，far ポインタと同様にセグメントアドレスとオフセットを持っています。しかし，far ポインタとは異なり，huge ポインタは**正規化**されており，「**far ポインタ**」で述べたような問題が解決されています。

正規化されたポインタとはどのようなものでしょうか。それは，セグメントアドレスを可能な限り大きくした32ビットのポインタです。セグメントの開始アドレスは16バイト（16進で10）ごとにとることができるので，正規化されたポインタのオフセットの値は，必ず0～15（16進で0～F）の範囲内にあることになります。

どのようにしてポインタを正規化するのでしょうか。簡単です。ポインタを20ビットアドレスに変換し，右の4ビットをオフセット，左の16ビットをセグメントアドレスに使えばいいのです。たとえば，2F84：0532というポインタは，2FD72という絶対アドレスに変換され，これが正規化されて2FD7：0002になります。以下に正規化の例をいくつか示します。

```
0000:0123  →  0012:0003
0040:0056  →  0045:0006
500D:9407  →  594D:0007
7418:D03F  →  811B:000F
```

このように，huge ポインタは常に正規化されています。なぜこれが重要なのでしょうか。どのようなメモリアドレスであっても，それに対応する huge アドレス（セグメント：オフセット）はただ1つだけだからです。したがって，==と!=の演算子は，huge ポインタに対しては正しい答えを返すことになります。

さらに，>，>=，<，<=は，huge ポインタに対しては，32ビット全部を使うようになって

います。正規化は，ここでも正しい答えを保証することになるわけです。

正規化によって，huge ポインタのオフセット値は，自動的に16ごとに0に戻り，セグメント値は（far ポインタとは異なり）正しく調整されます。たとえば，811B：000F をインクリメントすると，結果は811C：0000になります。同様に811C：0000をデクリメントすると，811B：000F になります。このため，huge ポインタを使えば，64K 以上の大きさのデータ構造を扱えるようになります。

huge ポインタを使う代償として，いくらかオーバーヘッドを必要とします。huge ポインタの演算は，特別なサブルーチンを呼んで行ないます。このため，huge ポインタの計算は，far や near ポインタにくらべ，かなりの時間がかかります。

Turbo C の 6 つのメモリモデル

オーバーヘッドを避けることは（それが必要なら別ですが），Turbo C によって可能になります。6つのメモリモデル（タイニィ，スモール，ミディアム，コンパクト，ラージ，ヒュージ）の中から，使用するメモリモデルを選択することができます。いプログラムが必要とするものによってどのモデルを選ぶかが決まります。以下は，各メモリモデルの要約です。

タイニィ　名前からもわかるように，最も小さいメモリモデルです。4つのセグメントレジスタ（CS,DS,SS,ES）がすべて同じアドレスにセットされるため，コード，データ，配列などの総計は64K 以内になります。常に near ポインタが使用されます。メモリが少ないときにはこれを使用してください。タイニィモデルのプログラムは，/t オプションをつけてリンクすれば，.COM 形式に変換することができます。

スモール　コードセグメントとデータセグメントは別々に，またオーバーラップせずにとられるので，コードと静的データはそれぞれ64K まで使用できます。スタックとエクストラセグメントは，データセグメントと同じアドレスに置かれます。near ポインタが常に使用されます。平均的なプログラム開発に適したモデルです。

ミディアム　コードには far ポインタが使用されます（データには near）。したがって，静的なデータは64K 以内に限定されますが，コードは1M バイトまでとることができます。大きなプログラムで，たくさんのデータをメモリ上に置いておく必要のない場合に向いています。

コンパクト　ミディアムの場合とは逆に，データに対してのみ far ポインタが使用されます。したがって，コードは64K 以内に制限されますが，データは最大1M バイ

トを利用することができます(ただし静的データは64Kまでです)。コードは小さいけれども，たくさんのデータをメモリ上に置くプログラムに適したモデルです。

ラージ　コードとデータの両方にfarポインタが使用され，コード･データとも1Mバイトまで使用できます(その合計は1Mバイトまでです)。非常に大きなアプリケーションでなければ必要にならないでしょう。

ヒュージ　コードとデータに対してfarポインタが使用されます。他のメモリモデルでは静的データのサイズは64Kに制限されますが，ヒュージメモリモデルではその制限はなくなり，64Kバイト以上の静的データを使用することができます。

以下に示した図（**図12.2〜図12.7**）は，Turbo Cの6つのメモリモデルにおいて8086のメモリがどのように区分されているかを示しています。

図12.2　タイニィモデルのメモリセグメンテーション

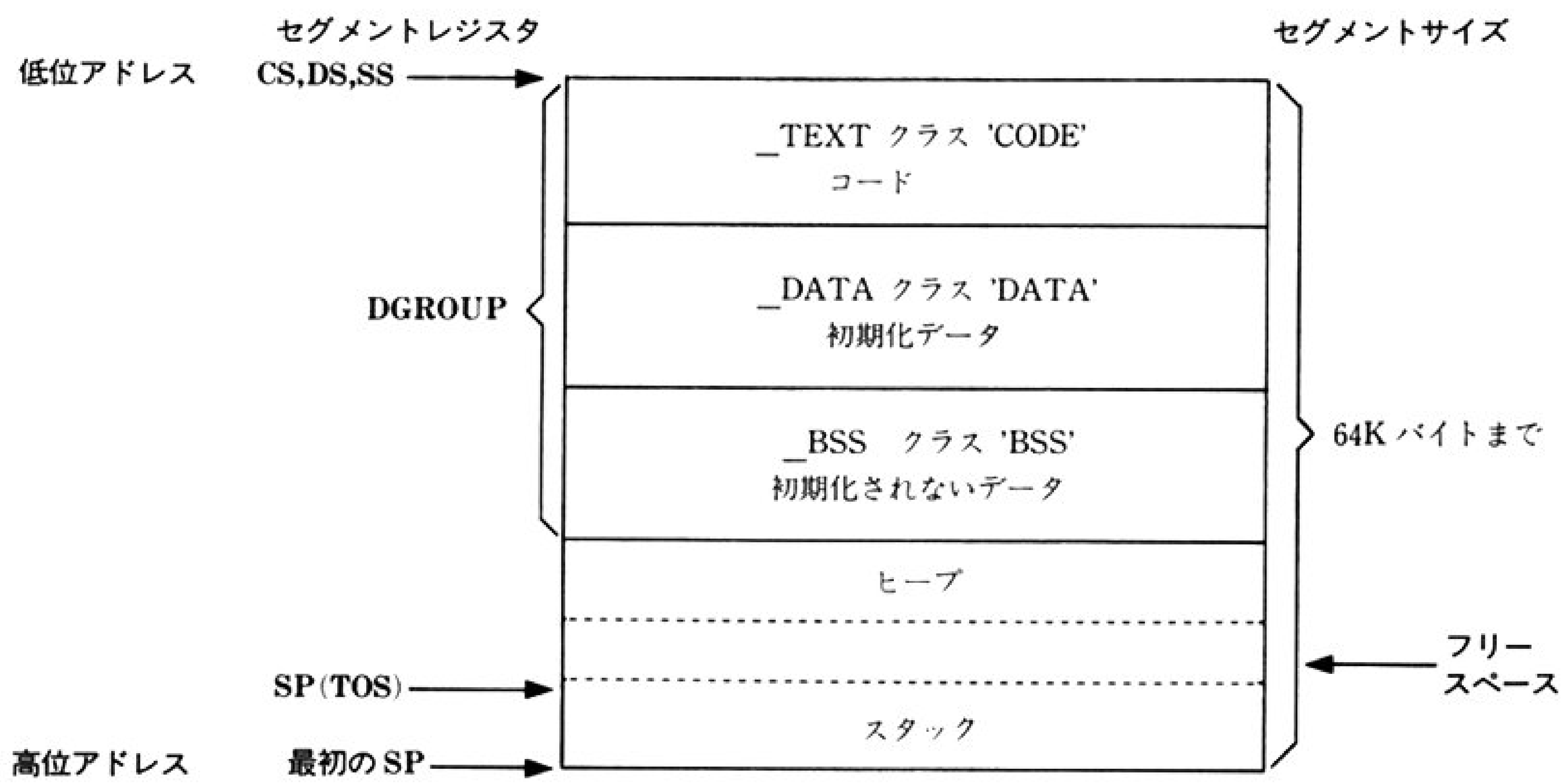


図 12.3　スモールモデルのメモリセグメンテーション

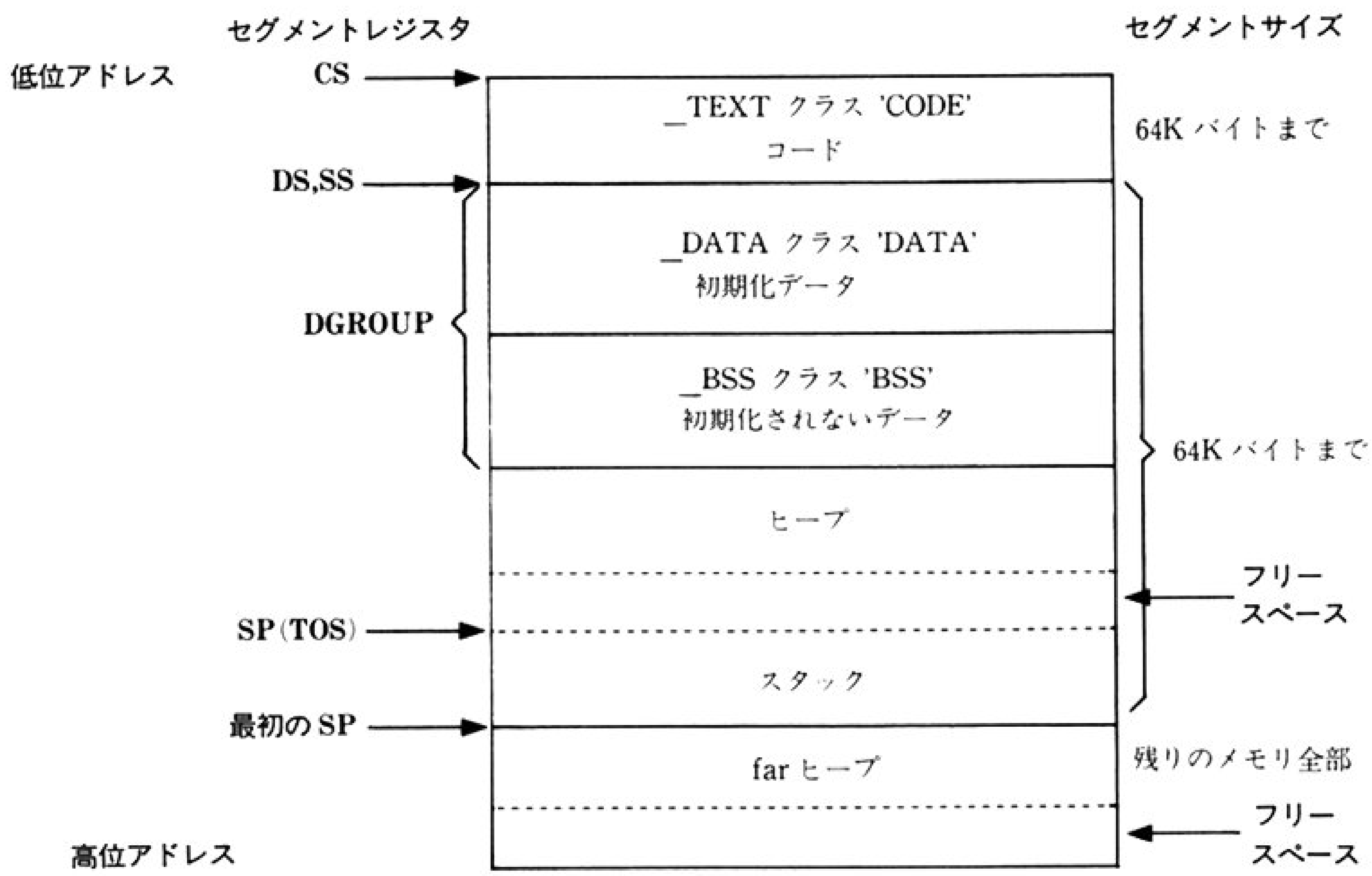


図 12.4　ミディアムモデルのメモリセグメンテーション

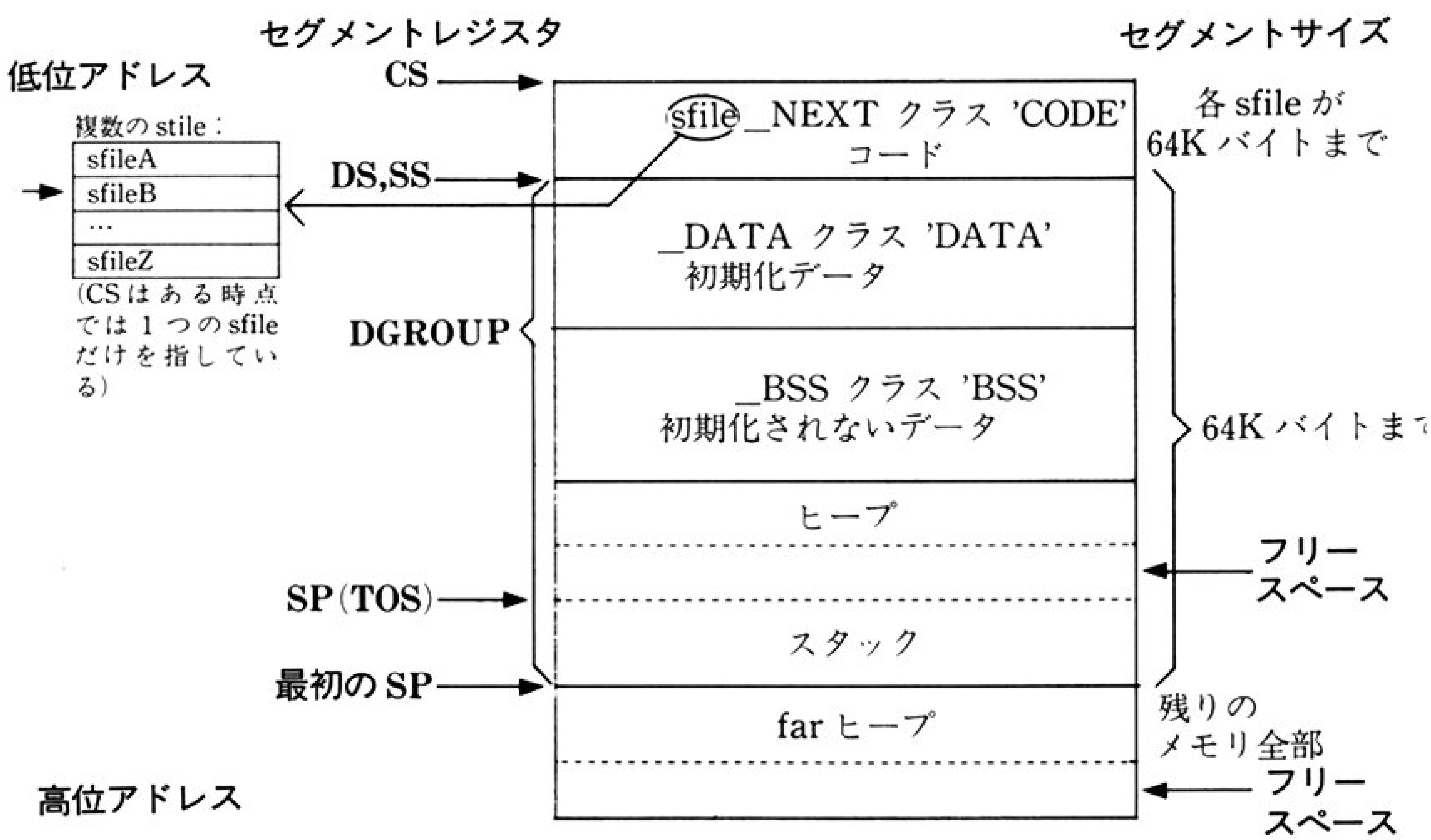


図 12.5　コンパクトモデルのメモリセグメンテーション

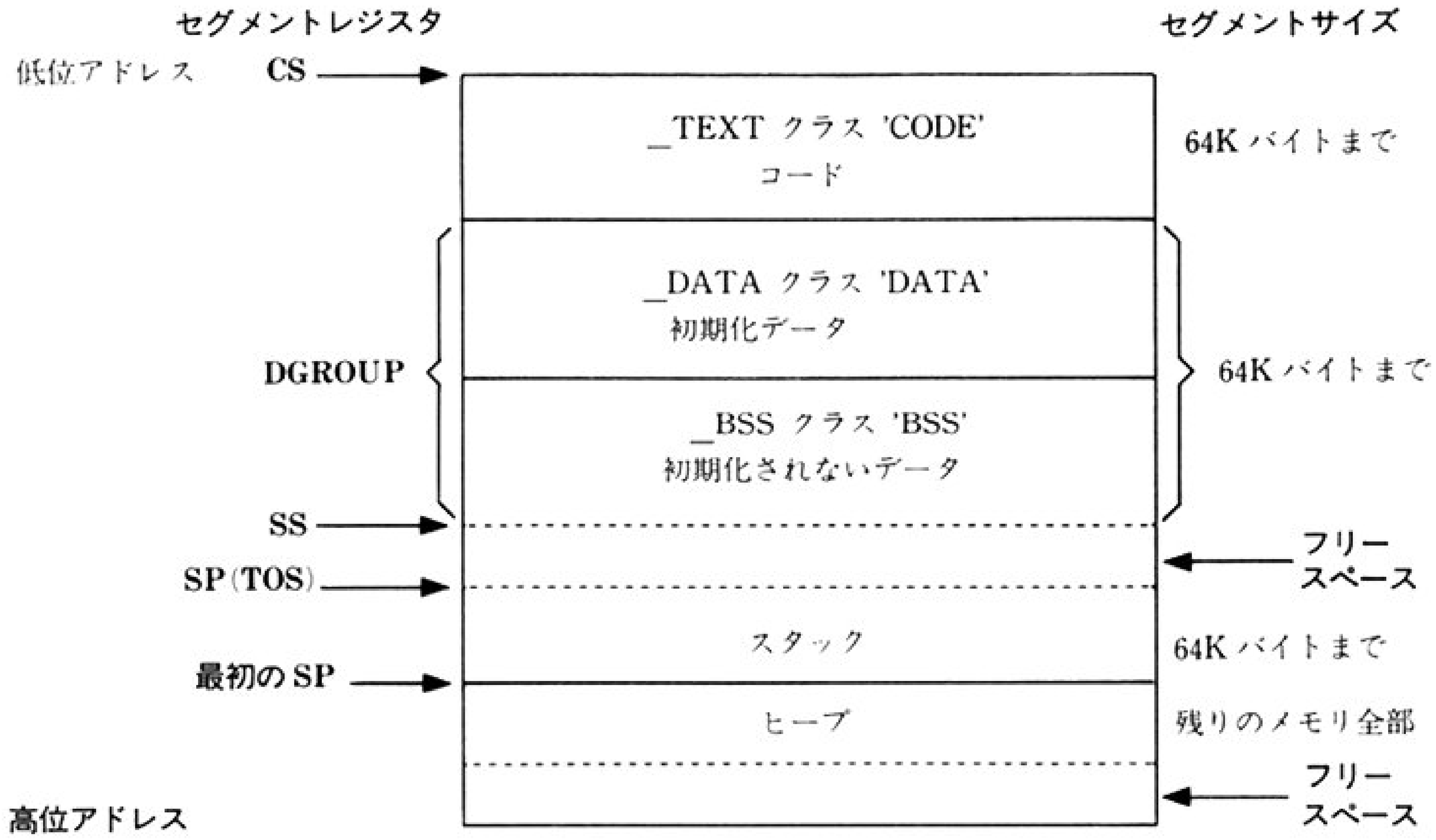


図 12.6　ラージモデルのメモリセグメンテーション

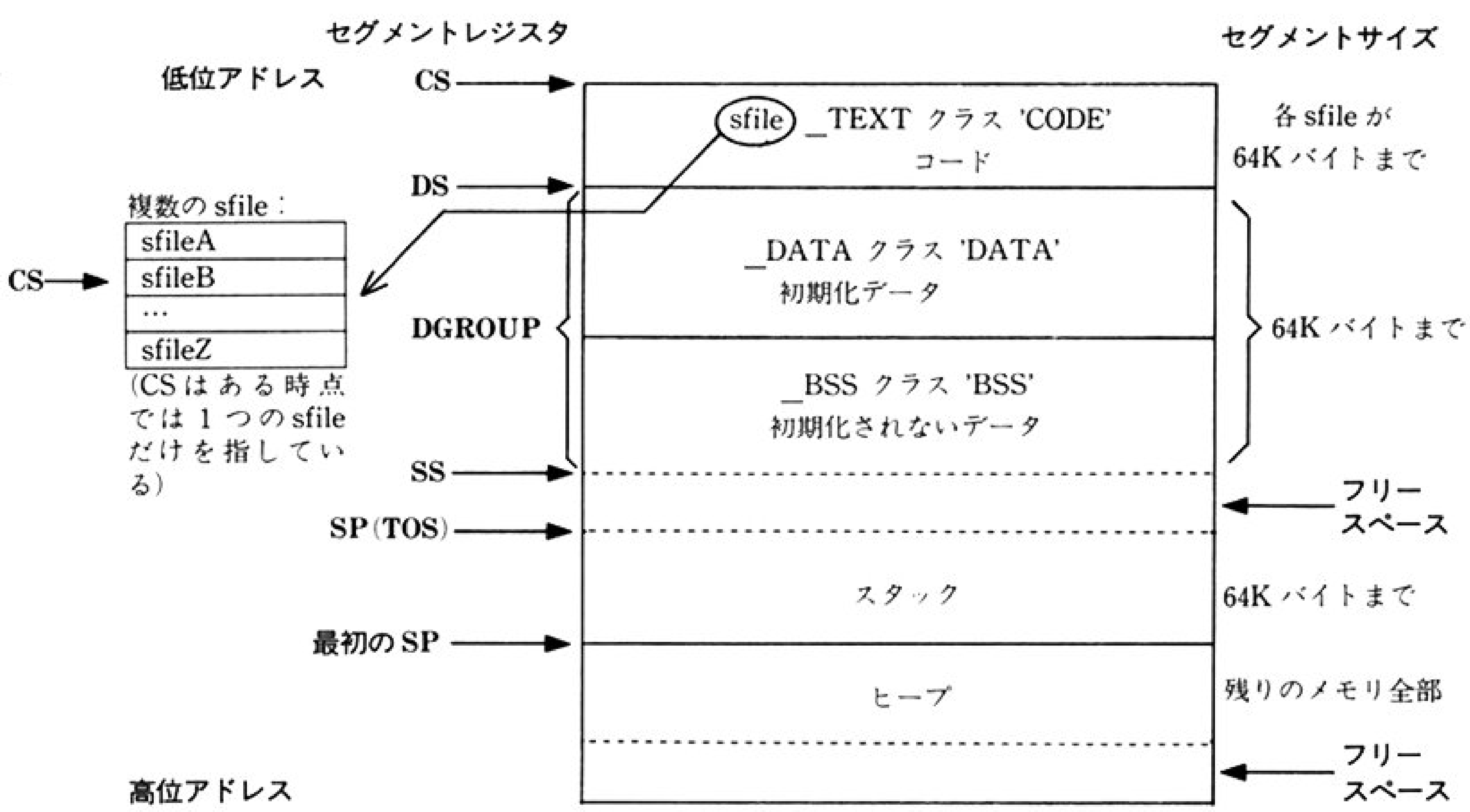


図 12.7　ヒュージモデルのメモリセグメンテーション

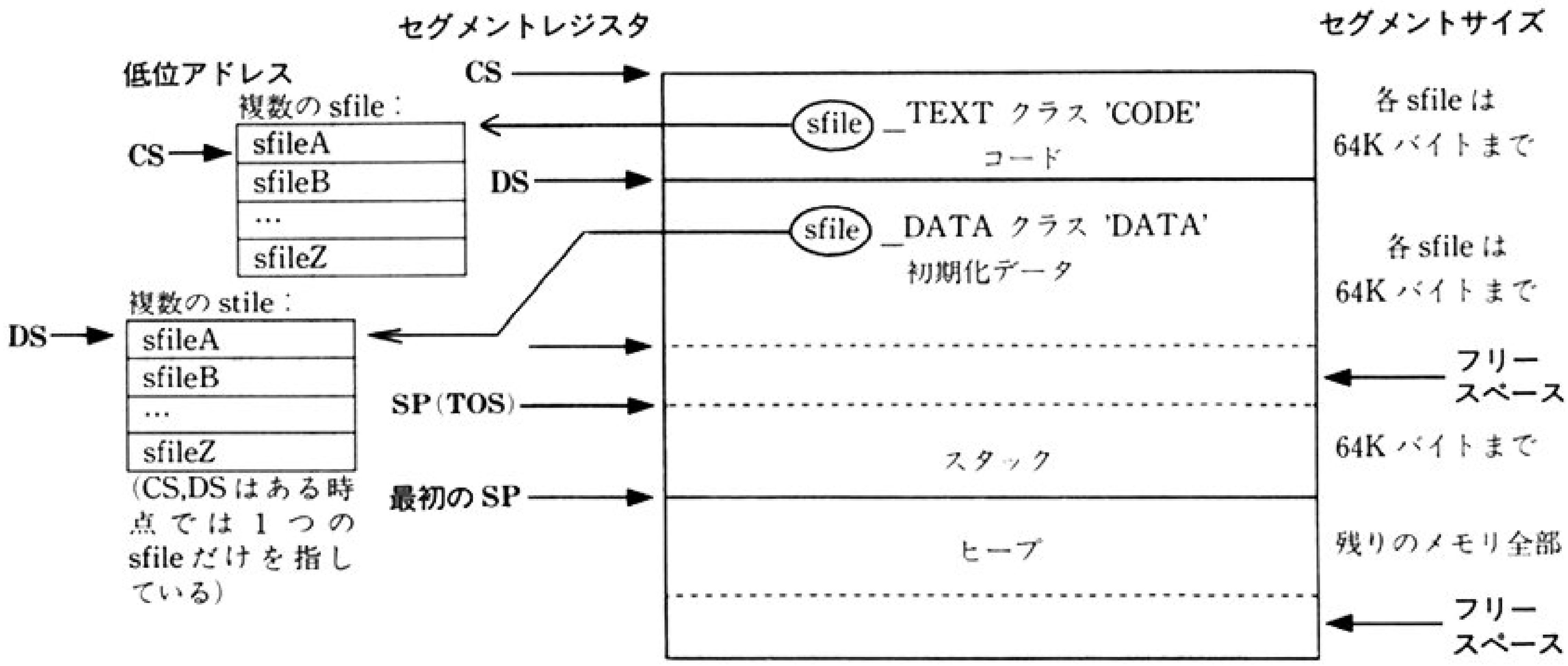


表 12.1 は，以上の6つのメモリモデルの違いを簡単にまとめたものです。各メモリモデルは，コードあるいはデータの大きさが，**スモール**（64Kb）か，**ラージ**（1Mb）かという観点から4つに分類することができます（**表 12.1** の行と列に対応します）。たとえば，タイニィ，スモール，コンパクトは，コードポインタのデフォルトは near なので，**スモールコードモデル**であると言えます。同様に，コンパクト，ラージ，ヒュージは，データポインタのデフォルトは far なので，**ラージデータモデル**ということになります。ヒュージモデルでも，データポインタのデフォルトは huge でなく far であるという点に注意してください。したがって，huge データポインタを使う場合には，明示的に huge と宣言する必要があります。

表 12.1　メモリモデル

データサイズ	コードサイズ 64K	コードサイズ 1MB
64K	タイニィ (データとコードはオーバーラップ 総計 ＝ 64K) スモール (オーバーラップはなし 総計 ＝ 128K)	 ミディアム (スモールデータ, ラージコード)
1MB	コンパクト (ラージデータ, スモールコード)	ラージ (ラージデータ, ラージコード) ヒュージ (ラージと同様, ただし 静的データ ＞ 64K)

重要な注意：1つのモジュール（いくつかのルーチンを含むソースファイル）をコンパイルする場合，コンパイル後のコードは64K バイトを越えてはなりません。1つのモジュールのコードは，1つのコードセグメントにおさめなければならないからです。これは，ラージコードモデル（ミディアム, ラージ, ヒュージ）を使用する場合でも同じです。1つのコードセグメント(64K)に入りきらないような大きなモジュールの場合は，64K 以内のモジュールに分割し，別々にコンパイルしてからリンクする必要があります。同様に，ヒュージモデルは，合計では64K 以上の静的データを扱えますが，各モジュール内では64K を越えてはなりません。

複数のメモリモデルを使ったプログラミング :アドレス修飾子

Turbo C は，標準の C（K&R あるいは ANSI）にはない7つの新しい予約語（near,far, huge,_cs,_ds,_es,_ss）を導入しています。これらは，ポインタの修飾子として（場合によっては関数の修飾子としても）使用することができます。ただし，いくつかの制限と注意点があります。

Turbo　C では，予約語 near,far,huge で関数やポインタを修飾することができます。near,far,huge の各データポインタについては，この章の初めの方で説明しました。near 関数は，near コールで呼び出され，near リターンで帰ります。同様に，far 関数は，far コールで呼び出され，far リターンで戻ります。huge 関数は far 関数と似ていますが，huge 関数は DS を新しい値にセットすることができるのに対し，far 関数はできません。

さらに，_cs,_ds,_ss,_es という4つの特殊な near データポインタがあります。これらは16ビットのポインタで，それぞれ名前に対応するセグメントレジスタに結びつけられています。たとえば，次のようなポインタを宣言すると，

```
char _ss *p;
```

p はスタックセグメントに対する16ビットオフセット値をとることになります。

あるプログラムの中の関数とポインタは，選択したメモリモデルによって，デフォルトとして near か far になります。関数あるいはポインタが near であれば，それは CS か DS レジスタのどちらかに結びつけられていることになります。

表 12.2 に，各メモリモデルのデフォルトポインタの種類を示しました。ポインタの大きさは，そのポインタが64K のメモリ内で働くか（near，セグメント内），それとも1M バイトのメモリ空間内で働くか（far，セグメントアドレスを持つ）に対応して決まることに注意してください。

表 12.2　デフォルトのポインタの型

メモリモデル	関数ポインタ	データポインタ
タイニィ	near, _cs	near, _ds
スモール	near, _cs	near, _ds
ミディアム	far	near, _ds
コンパクト	near, _cs	far
ラージ	far	far
ヒュージ	far	far

関数を near と宣言するか, far と宣言するか

場合によっては，**表 12.2** に示したメモリモデルから決まるデフォルトの関数の型を変更したい（あるいは変更する必要がある）ことがあります。

たとえば，ラージメモリモデルを使いたい場合に，プログラムの中に次のようなリカーシブ（再帰的——自分で自分自身を呼び出す）関数があるとします。

```
double power(double x, int exp)
{
    if (exp <= 0)
        return(0);
    else
        return(x*power(x,exp-1));
}
```

すると，*power* が自分自身を呼び出すたびに far コールをしなければなりませんが，far コールはスタックスペースとクロックサイクルをたくさん使ってしまいます。*power* を，次のように near 型と宣言すれば，すべての呼び出しが near コールになるので，こうしたオーバーヘッドを減らすことができます。

```
double near power(double x, int exp)
```

これで，*power* は，それがコンパイルされたコードセグメントの中でのみ呼び出すことができ，またすべての呼び出しは near コールとなることが保証されます。

これは，もしラージコードモデル（ミディアム，ラージ，ヒュージ）を使っている場合，*power* が定義されているモジュールの中でしか，*power* を呼び出せないことを意味します。

他のモジュールは各々独自のコードセグメントを持っているので，別のモジュールにあるnear関数を呼び出すことはできません。さらに，near関数は，それを呼び出す前に，定義または宣言しておかなければなりません。そうしないと，コンパイラはnearコールを生成する必要のあることがわからないからです。

逆に，関数をfarと宣言すれば，farコールとfarリターンが生成されます。スモールコードモデル（タイニィ，スモール，コンパクト）では，far関数は使用する前に宣言または定義して，確実にfarコールが行なわれるようにしなければなりません。

*power*の例をもう一度見てください。*power*がそのモジュールの中でしか呼び出されないとすれば，**static**と宣言した方がよいことがわかります。**static**と宣言すると，その名前はそのモジュールの外の関数からは“見えなくなる”からです。また，*power*の引数の個数は固定なので，次のように**pascal**型と宣言すれば，さらに最適化することができます。

```
static double near pascal power(double x, int exp)
```

ポインタをnear,far,hugeのいずれと宣言するか

関数をプログラムの他の部分とは異なるモデルとして宣言する理由を見てきました。ポインタについても同様ではないでしょうか。前の節とまったく同じ理由から，不要なオーバーヘッドを減らし（デフォルトがfarの際にnearと宣言する），デフォルトのセグメントの外にあるものを参照することができます（デフォルトがnearの際に**far**，**huge**と宣言する）。

もちろん，このようなことをする場合には“落し穴”がいくつかあります。たとえば，次のプログラムをスモールモデルでコンパイルしたとします。

```
void myputs(s)
char *s;
{
    int  i;
    for (i=0; s[i] != 0; i++) putc(s[i]);
}

main()
{
    char  near *mystr;

    mystr = "Hello, world¥n";
    myputs(mystr);
}
```

このプログラムは正しく動きます。すべてのポインタ，コードおよびデータはnearなので，*mystr*をnearと宣言するのも不要といえます。

しかし，このプログラムをコンパクト（あるいはラージ，ヒュージ）メモリモデルでコンパイルするとどうなるのでしょうか。**main** の *mystr* は near（16ビットポインタ）のままです。しかし，*myputs* の中のポインタ *s* は今度はデフォルトの far になります。したがって，*myputs* は far ポインタを作るためにスタックから2ワードを取り出そうとしますが，できあがったアドレスは *mystr* のものとは違っているはずです。

どうすればこの問題を避けることができるでしょうか。答えは，次のようにモダンスタイルで *myputs* を定義することです。

```
void myputs(char *s)
{
    /* myputs のボディ */
}
```

Turbo C は，これをコンパイルするときに，*myputs* は char へのポインタを期待しているとわかっています。また，ラージモデルでコンパイルしているので，ポインタが far でなければならないことがわかっています。したがって Turbo C は，*mystr* の16ビットポインタとともにデータセグメント（DS）レジスタをスタックにプッシュして，far ポインタを作り出すのです。

これと逆の場合はどうでしょうか。つまり，*myputs* を far と宣言し，スモールデータモデルでコンパイルする場合です。この場合も，関数プロトタイプを使わないと，**main** はオフセットとセグメントアドレス両方をスタックにプッシュしますが，*myputs* はオフセットしか期待していないので，問題が起きるはずです。関数プロトタイプを使えば，**main** はオフセットのみをスタックにプッシュすることになります。

教訓：ポインタの型を明示的に far または near と宣言する場合には，そのポインタを使用する関数は必らず関数プロトタイプを宣言すべきである。

特定のセグメント：オフセットアドレスを指し示す

far ポインタが，メモリ上のある位置（特定のセグメント：オフセットアドレス）を指し示すようにするにはどうすればよいでしょうか。標準のライブラリルーチンとして **MK_FP** が用意されています。**MK_FP** は，セグメント値とオフセット値を引数にとり，far ポインタを返します。たとえば次のようにします。

```
farptr = MK_FP(segment_value, offset_value);
```

また，far ポインタ（*fp*）から，セグメント値を取り出す **FP_SEG**(*fp*) と，オフセット値を取り出す **FP_OFF**(*fp*) も用意されています。この3つのライブラリルーチンの詳細については，リファレンスガイドの第2章を参照してください。

適切な宣言子を作る

宣言子というのは，関数，変数，ポインタ，データ型の宣言を行なう際に使用する文のことです。Cでは，複雑な宣言子を作ることも可能です。この節では，宣言子を書く（あるいは読む）練習になるような例をいくつか示し，あわせて犯しやすい誤りも紹介しています。

伝統的なCのプログラミングでは，必要な定義をネストして宣言子を作り上げなければなりません。こうすると，プログラムは読みにくく（また書きにくく）なってしまいます。

以下の例に示す宣言子を考えてみましょう（**表12.3**）。これらをスモールメモリモデル（コード・データともスモール）でコンパイルすることにします。

表12.3　typedefを使わない宣言子

`int f1();`	int を返す関数
`int *p1;`	int を指すポインタ
`int *f2();`	int を指すポインタを返す関数
`int far *p2;`	int を指す far ポインタ
`int far *f3();`	int を指す far ポインタを返す near 関数
`int *far f4();`	int を指す near ポインタを返す far 関数
`int (*fp1)(int);`	int を引数にとり， int を返す関数を指すポインタ
`int (*fp2)(int *ip);`	int を指すポインタを引数にとり， int を返す関数を指すポインタ
`int (far *fp3)(int far *ip);`	int を指す far ポインタを引数にとり， int を返す関数を指す far ポインタ
`int (far *list[5])(int far *ip);`	int を指すfarポインタを引数にとり， int を返す関数を指す far ポインタ5個の配列
`int (far *gopher(int(far *fp[5])(int far *ip)))(int far *ip);`	int を指すfarポインタを引数にとる int を返す関数を指す far ポインタ5個の配列を引数にとり， そのようなポインタを1つ返す near 関数

以上はすべて正しい宣言子ですが，徐々に理解するのが難しくなっていきます。しかし，typedefをうまく使うと，こうした宣言子を読みやすくすることができます。

同じ宣言子をtypedef文を使って書きなおすと，次のようになります。

表 12.4 typedef を使った宣言子

`int f1();`	int を返す関数
`typedef int *intptr;`	
`intptr p1;`	int を指すポインタ
`intptr f2();`	int を指すポインタを返す関数
`typedef int far *farptr;`	
`farptr p2;`	int を指す far ポインタ
`farptr f3();`	int を指す far ポインタを返す near 関数
`intptr far f4();`	int を指す near ポインタを返す far 関数
`typedef int (*fncptr1)(int);`	
`fncptr fp1;`	int を引数にとり，int を返す関数を指すポインタ
`typedef int(fncptr2)(intptr);`	
`fncptr2 fp2;`	int を指すポインタを引数にとり，int を返す関数を指すポインタ
`typedef int(far *ffptr)(farptr);`	
`ffptr fp3;`	int を指す far ポインタを引数にとり，int を返す関数を指す far ポインタ
`typedef ffptr ffplist[5];`	
`ffplist list;`	int を指す far ポインタを引数にとり，int を返す関数を指す far ポインタ5つの配列
`ffptr gopher(ffplist);`	int を指す far ポインタを引数にとり，int を返す関数を指す far ポインタ5つの配列を引数にとり，同様なポインタを1つ返す near 関数

見てわかるように，*gopher* の typedef 宣言と前に示した宣言とでは，読みやすさと明確さが大きく異なります。typedef と関数プロトタイプをうまく使えば，プログラムは書きやすく，デバッグしやすく，保守しやすくなるのです。

ライブラリを使うには

Turbo C には，6個のメモリモデルに対して，標準のライブラリルーチンがそれぞれ1つずつ用意されています。統合環境（TC）を使っている場合，Turbo C は選択されたメモリモデルに基づいて，適切なライブラリを適切な順序でリンクします。同様に，スタンドアロンのコンパイラ（TCC）を使う場合も，Turbo C は自動的にリンクを行ないます。

ただし，TLINK（Turbo C のリンカ）を直接（スタンドアロンのリンカとして）使用する場合は，どのライブラリを使うのかを指定しなくてはなりません。6個のメモリモデルすべてを使うのでなければ，（作業用ディスクやハードディスクに）使用するモデルのライブ

リファイルだけをコピーしておけばよいでしょう。各メモリモデルで必要となるライブラリファイルの一覧表を次に示します。

タイニィ	C0T.OBJ, MATHS.LIB, CS.LIB
スモール	C0S.OBJ, MATHS.LIB, CS.LIB
コンパクト	C0C.OBJ, MATHC.LIB, CC.LIB
ミディアム	C0M.OBJ, MATHM.LIB, CM.LIB
ラージ	C0L.OBJ, MATHL.LIB, CL.LIB
ヒュージ	C0H.OBJ, MATHH.LIB, CH.LIB

ここで，タイニィモデルとスモールモデルは，ライブラリファイルは同じものを使いますが，スタートアップファイルは異なる（C0T.OBJ，C0S.OBJ）ことに注意してください。また，使用しているシステムに8087/80287演算コプロセッサが装着されている場合は，FP87/80287.LIB というファイルも必要となります。8087/80287がなく，エミュレートする必要がある場合には，EMU.LIB が必要となります。

次に TLINK コマンドの入力例をいくつか示します。

```
tlink c0m a b c, prog, mprog, fp87 mathm cm
tlink c0c d e f, plan, mplan, emu mathc cc
```

最初のコマンドは，ミディアムモデルのライブラリと8087/80287サポートライブラリをリンクして，実行可能プログラム PROG.EXE を作成します。2番目は，コンパクトモデルのライブラリと，実行時にコプロセッサが使えない場合に浮動小数点演算をエミュレートするライブラリをリンクして，PLAN.EXE を作成します。

注意：オブジェクトとライブラリを並べる順序は，たいへん重要です。C スタートアップモジュール（C0*x*.OBJ）は，常にリストの先頭に置く必要があります。ライブラリは次の順序で並べます。

1. ユーザ独自のライブラリ（必要であれば）
2. FP87.LIB または EMU.LIB，次に MATH*x*.LIB（浮動小数点を扱う場合だけ必要）
3. C*x*.LIB（Turbo C の標準ライブラリファイル）

C0*x*, MATH*x*, C*x* の *x* は，メモリモデルを指定する英字で，T, S, M, C, L, H のいずれかです。

メモリモデルが異なるモジュールのリンク

あるモジュールはスモールメモリモデルでコンパイルし，他のモジュールはラージモデルでコンパイルしたような場合，リンクはどうすればよいでしょうか。

リンクそのものはうまくいくでしょう。しかし，ここで直面する問題は「**関数を near と宣言するか，far と宣言するか**」で述べたようなことです。スモールモジュール内の関数が，ラージモジュール内の関数を呼び出していると，near コールが行なわれるはずで，悲惨な結果を招くでしょう。ポインタに関しても，やはり「**ポインタを near,far,huge のいずれと宣言するか**」で述べたような問題にぶつかります。スモールモジュール内の関数は，near ポインタを受け渡すことを期待しているのに対し，ラージモジュール中の関数は far ポインタを期待しているわけです。

これを解決するには，やはり関数プロトタイプを使うことです。*myputs* を別のモジュールに移し，ラージメモリモデルでコンパイルしたとしましょう。このような場合，MYPUTS.H などの名前でヘッダファイルを作って，その中に次のような関数プロトタイプを入れておくべきでしょう。

```
void far myputs(char far *s);
```

さらに，main を独立したモジュールにするのであれば，その中でヘッダファイルをインクルードします。

```
#include <stdio.h>
#include "myputs.h"

main()
{
     char near *mystr;
     mystr = "Hello, world¥n";
     myputs(mystr);
}
```

このプログラムをコンパイルすると，Turbo C は，MYPUTS.H から関数プロトタイプを読み込み，*myputs* は far ポインタを引数にとる far 関数であると認識することができます。したがって，スモールメモリモデルでコンパイルしても，呼び出しのコードは適切に生成されるのです。

では，以上のことに加えて，ライブラリルーチンをリンクしなければならないとしたらどうでしょうか。最善の方法は，ラージモデルライブラリの1つを使って，すべてを far と宣言することです。これを行なうには，通常のヘッダファイル (STDIO.H など) をコピー

し，適当な名前（FSTDIO.H など）をつけておきます。そして，このファイルの中の各関数プロトタイプを，次のように明示的に far 型と宣言します。

```
int far cdecl printf(char far * format, ...);
```

こうすると，各ルーチンに対して，far コールが行なわれるだけでなく，渡される引数も far ポインタになります。プログラムを次のように変更して，新しいヘッダファイルをインクルードするようにしてください。

```
#include <fstdio.h>

main()
{
    char  near *mystr;

    mystr = "Hello, world¥n";
    printf(mystr);
}
```

これを TCC でコンパイルし，TLINK でリンクしてみてください。リンク時には，ラージモデルライブラリ CL.LIB を指定します。メモリモデルを組み合わせるのは，ややトリッキィですができないことではありません。しかし，なにか間違いをすれば，難解なバグが発生することは覚悟しておいてください。

複数の言語を使うプログラミング：他の言語とのインターフェース

Turbo C では，C のプログラムから，他の言語で書いたルーチンを容易に呼び出すことができます。また逆に，他の言語のプログラムから C のルーチンを呼び出すこともできます。この節では，Turbo C と他の言語とのインターフェースがどの程度簡単なのかを明らかにするとともに，こうしたインターフェースに関する情報も提供しています。

最初に，引数の受け渡しシーケンス（手順）として代表的な2つを説明し，次にアセンブリ言語モジュールの書き方を示すことにしましょう。

引数の受け渡しシーケンス：C と Pascal

Turbo C は，関数へ引数を受け渡す方法として2つをサポートしています。1つは標準の C の方法，もう1つは Pascal の方法です。

C の引数受け渡しシーケンス

まず次のような関数プロトタイプを宣言した場合を考えてみましょう。

```
void  funca(int p1, int p2, long int p3);
```

デフォルトでは，Turbo C は“C の引数受け渡しシーケンス”（C calling convention..C の呼び出し慣例とも呼ばれる）を用います。この関数（*funca*）が呼び出されると，引数は“右から左”の順（*p3, p2, p1* の順）でスタックにプッシュされていきます。その後，リターンアドレスがスタックにプッシュされます。したがって，次のような呼び出しを行なうと，

```
main()
{
     int   i,j;
     long  k;
     ...
     i = 5; j = 7; k = 0x1407AA;
     funca(i,j,k);
     ...
}
```

スタックは次のようになるはずです（リターンアドレスがプッシュされる前）。

```
SP + 06:   0014
SP + 04:   07AA   k = p3
SP + 02:   0007   j = p2
SP:        0005   i = p1
```

(8086では，スタックは高位メモリから低位メモリに向かって成長していくことを思いだしてください。したがって，*i* がスタックの先頭になります)。呼び出されているルーチンは，スタックに何個の引数がプッシュされているかを正確に知っている必要はありません(知ることはできませんが)。そのルーチンとしては，必要な引数はスタックにあるはずだと想定しているだけです。

さらに，これが重要なポイントですが，呼び出されたルーチンは，スタックから引数をポップしてはなりません。なぜでしょうか。呼び出した側のルーチンがその作業を行なうからです。たとえば，上の **main** のソースからコンパイラが作り出すアセンブリ言語は，次のようなものになるでしょう。

```
mov   word ptr [bp-8],5        ; i = 5 をセット
mov   word ptr [bp-6],7        ; j = 7 をセット
mov   word ptr [bp-2],0014h    ; k = 0x1407AA をセット
mov   word ptr [bp-4],07AAh
push  word ptr [bp-2]          ; k の高位ワードをプッシュ
push  word ptr [bp-4]          ; k の低位ワードをプッシュ
push  word ptr [bp-6]          ; j をプッシュ
push  word ptr [bp-8]          ; i をプッシュ
call  near ptr funca           ; funca を呼び出す(アドレスをプッシュ)
add   sp,8                     ; スタックの調整
```

最後の命令 add sp,8に注目してください。コンパイラはこの段階で，スタックに何個の引数がプッシュされているかを正確に知っています。また，*funca* の呼び出しでプッシュされたリターンアドレスは，*funca* の最後の ret 命令によってすでにポップされていることも知っているわけです。

Pascalの引数受け渡しシーケンス

もう1つのやり方は，標準 Pascal の引数受け渡しシーケンス(Pascal の呼び出し慣例)です。ただしこれは，Turbo C から Turbo Pascal の関数が呼び出せることを意味するわけではありません。このシーケンスでは，引数を“左から右”へプッシュします。たとえば，*funca* を次のように宣言したとします。

```
void  pascal funca(int p1, int p2, long int p3);
```

この関数が呼び出されると，引数はスタックに左から右へ(*p1,p2,p3* の順で)プッシュされ，続いてリターンアドレスがスタックにプッシュされます。したがって次の呼び出しで

は,

```
main()
{
    int   i,j;
    long  k;
    ...
    i = 5; j = 7; k = 0x1407AA;
    funca(i,j,k);
    ...
}
```

スタックは次のようになります（リターンアドレスがプッシュされる前)。

```
SP + 06:   0005   i = p1
SP + 04:   0007   j = p2
SP + 02:   0014
SP:        07AA   k = p3
```

さて，どこが違うのでしょうか。引数をプッシュする順序が逆になっていることに加えて，Pascal のシーケンスでは，呼び出されるルーチン (*funca*) がスタック上にプッシュされている引数の個数を知っており，それにあわせてスタックを調整することができます。アセンブリコードにおきかえると，*funca* の呼び出しは次のようになります。

```
push  word  ptr [bp-8]        ; i をプッシュ
push  word  ptr [bp-6]        ; j をプッシュ
push  word  ptr [bp-2]        ; k の高位ワードをプッシュ
push  word  ptr [bp-4]        ; k の低位ワードをプッシュ
call  near  ptr funca         ; funca を呼び出す（アドレスをプッシュ）
```

呼び出しの後に add sp,8が無いことに注意してください。そのかわり，*funca* は **main** に戻る前に，スタックをきれいにするために ret 8という命令を実行します。

デフォルトでは，Turbo C で書かれる関数は，すべて C の引数受け渡し手順を使用します。唯一の例外は -p オプションを指定した場合で (**O/C/C/Calling convention.. Pascal**)，すべて Pascal の引数受け渡しシーケンスが使用されます。この場合でも，特定の関数に対して C のシーケンスを使うことが指定できます。これには，次のように修飾子 cdecl を使います。

```
void  cdecl funca(int p1, int p2, long int p3);
```

こうすると，その関数については-p オプションを無効にすることができます。

では，Pascal の呼び出し慣例はどんな場合に使うのでしょうか。主な理由は3つあります。

■ Pascal の呼び出し慣例を使った既存のアセンブリ言語ルーチンを呼び出すことがで

きる。

■他の言語で書かれたルーチンを呼び出すことができる。

■スタックをきれいにする命令を置く必要がないので，呼び出しのコードがいくらか小さくなる。

Pascalの呼び出し慣例を使用すると，どんな問題が起こってくるでしょうか。

まず第一に，Cの呼び出し慣例ほど強力ではありません。Cの場合のように，引数の個数を可変にすることはできません。呼び出されたルーチンは，渡された引数の個数を知って，それにあわせてスタックをきれいにしなければならないからです。引数が少なすぎても多すぎても重大な問題につながりますが，Cの慣例の場合は致命的とはなりません(間違った答えになることはあるでしょうが)。

第二は，-pオプションを使用する場合には，(プログラムで使っている)Cの標準関数に対するヘッダファイルをインクルードしておかなければなりません。そうしないと，標準関数は**Pascal**の慣例で呼び出され，プログラムは確実にクラッシュします。なぜなら，**1)** 引数が誤った順で渡され，**2)** スタックを誰もきれいにしないからです。

ヘッダファイルでは，各関数は**cdecl**と宣言されているので，これをインクルードしておけば，コンパイラはその宣言をみて，Cの呼び出し慣例を適用することができます。しかし，**cdecl**識別名は下線（_）がつけられるのに対し，**pascal**識別名にはつけられないので，-u-オプション（**O/C/C/**Generate underbars..Off）を指定しなければなりません。また，大文字小文字を区別しないリンク（**O/L/**Case sensitive link..Off）を行なう必要があります。この2つを忘れると，リンクエラーがたくさん出て，まさに大きなトラブルにまきこまれることになります。

まとめると以下のようになります。Turbo Cのプログラムの中でPascalの呼び出し慣例を使いたい場合は，できる限り関数プロトタイプを使うように心がけ，**cdecl**か**pascal**かを明示的に指定してください。またこの場合には，プロトタイプが必要であるという警告を出すオプションをオンにして(**O/C/E/L/F**：...On)，関数プロトタイプの宣言を忘れていないかどうか確認できるようにしておくのがよいでしょう。

アセンブリ言語インターフェース

2つの呼び出し慣例がどのように動くかを説明し，コンパイラが何をするのかお話してきました。では呼び出されるルーチンはどうすればよいでしょうか。今度は，Turbo Cから呼び出すことのできるアセンブリ言語ルーチンの書き方を見ていきましょう。

注意：この節は，8086アセンブリ言語ルーチンの書き方，セグメントやデータ定数の定義のしかたについてはすでにおわかりの方を対象としています。そうでない方は，Turbo アセンブラのユーザーズガイドとリファレンスガイド（あるいは8086に関する解説書）をご用意ください。

Turbo C から.ASM を呼び出すための準備

Cのプログラムにリンクするアセンブリ言語ルーチンは，モジュールとして書く必要があります。その際に，従わなければならない慣例があります。

- リンカが必要とする情報を与えること
- Cのプログラムのメモリモデルと，ファイルフォーマットが調和していること

アセンブリ言語ルーチンの一般的な姿は次のようになります。

```
識別名      名前          ファイル名

<code>      SEGMENT       BYTE PUBLIC 'CODE'
            ASSUME        CS;<code>, DS:<dseg>
            <........ コードセグメント .......>
<code>      ENDS

<dseg>      GROUP         _DATA, _BSS
<data>      SEGMENT       WORD PUBLIC 'DATA'
            <..初期化されるデータセグメント...>
<data>      ENDS

_BSS        SEGMENT       WORD PUBLIC 'BSS'
            <..初期化されないデータセグメント.>
_BSS        ENDS

            END
```

このレイアウトの中の，〈*code*〉，〈*data*〉，〈*dseg*〉は，使用するメモリモデルに応じて，特定の識別名に置きかえます。**表 12.5** に，各メモリモデルで，何に置きかえるのかを示してあります。表の中の *filename* はモジュールの名前で，NAME 命令と，識別名の置きかえの両方に使用しなければなりません。

ヒュージメモリモデルでは_BSS セグメントは存在せず，GROUP 定義が完全に落とされていることに注意してください。一般に_BSS はオプションであり，必要な場合にだけ定義します。

アセンブリ言語コードの“ひながた”を作る最良の方法は，何も行なわない‘カラ’のプログラムを（TCCの-Sオプションで）コンパイルして，.ASMを出力し，生成されたアセンブリコードを検討することです。

表 12.5　識別名の置きかえとメモリモデル

モデル	識別名の置きかえ	コードおよびデータポインタ
タイニィ，スモール	<code>= _TEXT <data>= _DATA <dseg>= DGROUP	Code: DW _TEXT:xxx Data: DW DGROUP:xxx
コンパクト	<code>= _TEXT <data>= _DATA <dseg>= DGROUP	Code: DW _TEXT:xxx Data: DD DGROUP:xxx
ミディアム	<code>= filename_TEXT <data>= _DATA <dseg>= DGROUP	Code: DD xxx Data: DW DGROUP:xxx
ラージ	<code>= filename_TEXT <data>= _DATA <dseg>= DGROUP	Code: DD xxx Data: DD DGROUP:xxx
ヒュージ	<code>= filename_TEXT <data>= filename_DATA <dseg>= filename_DATA	Code: DD xxx Data: DD xxx

データ定数とデータ変数の定義

メモリモデルは，データ定数（コード，データ，あるいはその両方を指すポインタ）をどのように定義するかにも影響を与えます。**表 12.5** を見れば，このポインタをどのように定義すべきかがわかります。*xxx* はポインタが指すアドレスです。

メモリモデルによって，DW（Define Word）と，DD（Define Doubleword）を使いわけていることに注意してください。これは，ポインタの大きさを示しています。数値やテキストの定数は，通常どおり定義します。

変数も，定数と同じように定義します。特定の値で初期化しない変数は，_BSSセグメントの中で宣言し，通常は値を書くところに疑問符（?）を置きます。

グローバルおよび外部識別名の定義

さて，モジュールの作成がすんだら，今度は Turbo C のプログラムに，どんな関数を呼び出すのか，またどんな変数を参照するのかを教えてやらなければなりません。アセンブリ言語ルーチンの中から，Turbo C の関数を呼び出したり，Turbo C プログラムの中で宣言された変数を参照したい場合も同じことが言えます。

こうした呼び出しを行なうには，Turbo C のコンパイラとリンカについて少し知っておかなければならないことがあります。外部識別名を宣言すると，コンパイラは自動的に下線（_）を識別名の前につけます。したがって，C のプログラムの中から参照しようとするアセンブリ言語モジュール内の識別名の前には下線をつけなくてはならないことになります。**pascal** 型の識別名は，C の識別名とは別の扱いを受け，すべて大文字にされ，その前には下線はつきません。

C の識別名につく下線はオプションで，デフォルトではオン（つける）になっていますが，-u-コンパイラオプションでオフにすることができます。しかし，標準の Turbo C ライブラリを使っていて，ライブラリを再構築するのでなければ，たくさんの問題に直面することになります（ライブラリの再構築には，実行時ライブラリのソースコードが必要となります。詳しいことは MSA にお問い合わせください）。

ソースファイルの中の **asm** コードが C 識別名（データあるいは関数）を参照している場合，参照される識別名は下線文字で始まっていなければなりません。

Turbo アセンブラ(TASM)は大文字/小文字の区別をしません。つまり，プログラムをアセンブルすると，識別名はすべて大文字になってしまいます。TASM の/mx スイッチを指定すると，外部名に関しては大文字小文字の区別をするようになります。Turbo C のリンカも，**extern** 識別名に関しては同じことを行なうので，万事うまくいくわけです。この章の例では，キーワードと指令は大文字で，他の識別名やオペコードは小文字で表わしているのに気づいた人もいるでしょう。この記述のしかたは，TASM のユーザーズガイドのスタイルに合わせたものですが，すべて大文字で(またはすべて小文字で)，または大文字小文字をおりまぜて記述してもかまいません。

特定の識別名（ルーチンまたは変数の名前）を，アセンブリ言語モジュールの外側から“見える”ようにするには，PUBLIC と宣言しなくてはなりません。

たとえば，*max* と *min* という整数型関数，および整数型変数 *MAXINT*, *lastmax*, *lastmin* を含むモジュールを書く場合，次の文をコードセグメントに置き，

```
PUBLIC  _max, _min
```

また，以下の文をデータセグメントに置きます。

```
PBULIC    _MAXINT, _lastmax, _lastmin
_MAXINT   DW  32767
_lastmin  DW  0
_lastmax  DW  0
```

.ASM から Turbo C ルーチンを呼び出すための準備

アセンブリ言語モジュールが，Turbo C のプログラムの中で宣言された関数や変数を参照できるようにするためには，EXTRN 文を使います。

関数を参照する

アセンブリ言語ルーチンから C 関数を呼び出せるようにするには，モジュールの中に次のような文を置きます。

```
EXTRN  関数名: near または far
```

コロンの後ろに，その関数が near か far かを指定します。near の場合には，EXTRN 文はモジュールのコードセグメント内に置き，far の場合は，EXTRN 文はすべてのセグメントの外側に置かれなければなりません。したがって，次の文をコードセグメント内に置くと，

```
EXTRN  _myCfunc1:near, _myCfunc2:near
```

アセンブリ言語ルーチンから *myCfunc*1と *myCfunc*2を呼び出すことができます。

データを参照する

変数を参照するには，データセグメントの中に，次の形式の EXTRN 文を置きます。

```
EXTRN  変数名 : サイズ
```

サイズには，変数の大きさを表わす以下の識別名のいずれかを指定します。

■ BYTE (1バイト)
■ WORD (2バイト)
■ DWORD (4バイト)
■ QWORD (8バイト)

■ TBYTE（10バイト）

配列の場合は，配列要素の大きさをサイズとして指定します。構造体の場合は，最も頻繁に使用される大きさをサイズとして指定します。

Cのプログラムの中で次のようなグローバル変数が宣言されている場合，

```
int   i,jarray[10];
char  ch;
long  result;
```

次の EXTRN 文をモジュール中に置けば，これらの変数が“見える”ようになります。

```
EXTRN  _i:WORD, _jarray:WORD, _ch:BYTE, _result:DWORD
```

最後の重要な注意：ヒュージメモリモデルを使う場合，EXTRN 文はすべてのセグメントの外側に置かなくてはなりません。これは，関数，変数どちらに対しても適用されます。

アセンブリ言語ルーチンの定義

これですべての準備はととのったので，実際にアセンブリ言語で関数をどのように書くかを見ていきましょう。そのためには，少し考えておく必要のある点がいくつかあります。引数の渡し方，値の返し方，レジスタに関する慣例の3つです。

C で次のようなプロトタイプが宣言されている関数 *min* を書きたいとしましょう。

```
int  extern  min(int v1, int v2);
```

min は，受け取った2つの整数値のうちの小さい方を返すことになっています。大枠は，次のようになります。

```
      PUBLIC  _min
_min  PROC    near
      .
      .
      .
_min  ENDP
```

もちろんここでは，*min* を near 関数とみなしているので，far 関数の場合は near を far に変更します。*min* の前に下線がつけられている点に注目してください。これで Turbo C のリンカが正しく参照関係を結びつけることができるのです。

引数の渡し方

最初に決めなければならないのは，引数受け渡し慣例はどれを使うかということです。Pascal の慣例を使う特別な理由がなければ，C の慣例ということになるでしょう。C の慣例であれば，*min* が呼び出されると，スタックは次のようになります。

```
SP + 04:    v2
SP + 02:    v1
SP:         リターンアドレス
```

ここでは，スタックをポップせずに引数を得たいので，ベースポインタ（BP）をセーブして，スタックポインタ（SP）の値をベースレジスタに格納した後，スタックポインタを直接スタックへのインデックスにして，目的の値を得ます。スタックに BP をプッシュすると，引数の相対オフセットは2だけ増すことに注意してください。スタックは2バイト増えているからです。

値の返し方

関数が整数値を返すとすると，返す値はどこに置けばよいのでしょうか。16ビット値（char, short, int, enum, near ポインタ）は AX レジスタに置きます。32ビット値（far, huge ポインタも含めて）は，DX も使って，高位ワード（ポインタのときはセグメントアドレス）は DX に，低位ワードは AX に置くことになっています。

float, double, long double の値は，8087/80287のスタックトップ（TOS）レジスタ ST(0) に置きます。8087/80287エミュレーションを使う場合には，エミュレータの TOS レジスタに置きます。

構造体の場合は，その値を静的なデータ領域に置き，その位置を指すポインタを返します（スモールデータモデルでは AX に，ラージデータモデルでは DX：AX に置きます）。

これを呼び出した関数は，その値を必要な場所にコピーしなければなりません。1～2バイトの構造体は（通常の int 型と同様に）AX に返され，4バイトの構造体の場合は AX：DX に返されます。

min の例では，16ビット値を扱っているので答えを AX に置くことになり，コードは次のようになります。

```
        PUBLIC  _min
_min    PROC    near
        push    bp              ; BP をスタックにセーブ
        mov     bp,sp           ; SP を BP にコピー
        mov     ax,[bp+4]       ; v1 を AX に置く
        cmp     ax,[bp+6]       ; v2 と比較し
        jle     exit            ; v1 > v2 ならば
        mov     ax,[bp+6]       ; v2 を AX にロード
exit:   pop     bp              ; BP をリストアして
        ret                     ; C に戻る
_min    ENDP
```

min を far 関数と宣言する場合はどうでしょうか。どんな変更が必要になるでしょうか。大きな違いは，関数に入ったときのスタックの状態です。

```
SP + 06:   v2
SP + 04:   v1
SP + 02:   リターンセグメント
SP         リターンオフセット
```

これで，オフセットはさらに2だけ増えることになります。2バイト（リターンセグメントの分）余計にスタックにプッシュされたからです。far 版の *min* は次のようになります。

```
        PUBLIC  _min
_min    PROC    far
        push    bp              ; BP をスタックにセーブ
        mov     bp,sp           ; SP を BP にコピー
        mov     ax,[bp+6]       ; v1 を AX に置く
        cmp     ax,[bp+8]       ; v2 と比較し
        jle     exit            ; v1 > v2 ならば
        mov     ax,[bp+6]       ; v2 を AX にロード
exit:   pop     bp              ; BP をリストアして
        ret                     ; C に戻る
_min    ENDP
```

ここで，*v1* と *v2* のオフセットは，スタック上の付加された2バイトを反映して，それぞれ2だけ増えていることに注意してください。

では，何かの理由で，*min* を pascal 関数と宣言する場合，つまり Pascal の引数受け渡しシーケンスを使う場合はどうなるでしょうか。関数に入ったときのスタックは次のようになります（*min* は near 関数にしています）。

```
SP + 04:   v1
SP + 02:   v2
SP :       リターンアドレス
```

C の慣例の場合に加えて，識別名 *min* に対して Pascal の慣例（大文字で下線なし）を適用しなくてはなりません。

この慣例では，*v1* と *v2* の位置が入れかわり，さらに *min* の実行が終わって呼び出しル

ーチンに返る際に，スタックをきれいにしなくてはなりません。それには，RET 命令に何バイト分ポップするかを指定します。ここでは，*v1* と *v2* が占めている4バイト（リターンアドレスは RET によって自動的にポップされます）をポップする必要があります。

改訂版は次のようになります。

```
          PUBLIC  MIN
MIN       PROC    near              ; Pascalバージョン
          push    bp                ; BP をスタックにセーブ
          mov     bp,sp             ; SP を BP にコピー
          mov     ax,[bp+6]         ; v1 を AX に置く
          cmp     ax,[bp+4]         ; v2 と比較し
          jle     exit              ; v1 > v2 ならば
          mov     ax,[bp+4]         ; v2 を AX にロード
exit:     pop     bp                ; BP をリストアし
          ret     4                 ; スタックをクリア
MIN       ENDP
```

最後の例は，C の引数受け渡しシーケンスの方が好まれる理由を示しています。*min* を次のように宣言した場合を考えてみましょう。

```
int  extern  min(int count,int v1,int v2,...);
```

min はいくつでも整数を受け入れ，その中の最小値を返すことになります。ただ，*min* としては引数の個数を自動的に知る手段はないので，最初の引数には個数を指定することにします。

たとえば，この関数は次のように使うことができるわけです。

```
i = min(5, j, limit, indx, lcount, 0);
```

ここで，*i*, *j*, *limit*, *indx*, *lcount* はすべて int 型（あるいはコンパチブルな型）であるとします。この関数に入ったときのスタックは次のようになるでしょう。

```
SP + 08:   (etc.)
SP + 06:   v2
SP + 04:   v1
SP + 02:   count
SP:        リターンアドレス
```

min の改訂版は次のようになります。

```
        PUBLIC  _min
_min    PROC    near
        push    bp              ; BP をスタックにセーブ
        mov     bp,sp           ; SP を BP にコピー
        mov     ax,0            ; AX = 0 をセット
        mov     cx,[bp+4]       ; count を CX に置いて
        cmp     cx,ax           ; 0 と比較し
        jle     exit            ; CX <= 0 ならば終了
        mov     ax,[bp+4]       ; 最初の値を AX に置いて
        jmp     ltest           ; 比較のループに入る
compare: cmp    ax,[bp+6]       ; 次の値と比較し
        jle     ltest           ; 次の値が小さければ
        mov     ax,[bp+6]       ; 次の値を AX にロードする
ltest:  add     bp,2            ; BP を次の値に合わせて
        loop    compare         ; ループの頭に戻る
exit:   pop     bp              ; BP をリストアし
        ret                     ; C に戻る
_min    ENDP
```

このプログラムは，*count* がどんな値をとっても正しく動くことに注意してください。

■ *count*<=0の場合は，0を返す。

■ *count*=1 の場合は，最初の値を返す。

■ *count*>=2の場合は，引数リストの中の値を順に比較し，最小値を返す。

レジスタに関する慣例

これまで，*min* の中でいくつかのレジスタを使ってきましたが，問題は無かったでしょうか。Turbo C のプログラムが使っているかもしれないレジスタについてはどうでしょうか。

心配はいりません。この関数は正しく書かれています。使用したレジスタのうち，一つだけ注意しなければならないのは BP で，関数に入ったときにスタック上に退避（プッシュ）し，関数の終わりでスタックから復帰させる（ポップする）必要があります。

この他に注意しなければならないレジスタは，SI と DI の2つです。この2つは，Turbo C がレジスタ変数に使用します。アセンブリ言語ルーチンの中で使用する場合は，ルーチンに入った段階で（おそらくスタックに）退避し，ルーチンの最後で回復する必要があります。しかし，-r-オプション（**O/C/O/Use register variables..Off**）で Turbo C のプログラムをコンパイルする場合は，SI と DI について心配する必要はありません（-r-オプションを使う際には注意事項があります。このレジスタ変数オプションの詳細については付録 B を参照してください）。

CS,DS,SS,ES の各レジスタは，使用しているメモリモデルに対応する値をとっています。その関係を次に示します。

タイニィ	CS=DS=SS ; ES=スクラッチ
スモール, ミディアム	CS!=DS, DS=SS ; ES=スクラッチ
コンパクト, ラージ	CS!=DS!=SS ; ES=スクラッチ (モジュールごとに CS が1つ)
ヒュージ	CS!=DS!=SS ; ES=スクラッチ (モジュールごとに CS と DS が1つずつ)

.ASM ルーチンから C の関数を呼び出す

もう，逆の場合についてもだいたいおわかりでしょう。アセンブリ言語モジュールから C の関数を呼び出すことができます。そのためには，まず C の関数をアセンブリ言語モジュールから“見える”ようにしなければなりません。すでに簡単に述べましたが, near か far 修飾子をつけて EXTRN を宣言します。たとえば，次のような C の関数を書いたとします。

```
long docalc(int *fact1, int fact2, int fact3);
```

わかりやすいように，*docalc* は（Pascal 関数ではなく）C 関数であることにしましょう。タイニィ，スモール，コンパクトのいずれかのメモリモデルを使うのであれば，アセンブリ言語モジュールの中の宣言は次のようになります。

```
EXTRN   _docalc:near
```

また，ミディアム，ラージ，ヒュージの場合は次のように宣言します。

```
EXTRN   _docalc:far
```

docalc は，3つの引数を指定して呼び出されます。

- *xval* という名前の記憶場所のアドレス
- *imax* という名前の記憶場所に置かれている値
- 421（10進）という定数値

さらに, 計算の結果は, *ans* と名づけられた32ビットの記憶場所に計算結果は格納されるとします。C で同様な呼び出しを表わしてみると，次のようになります。

```
ans = docalc(&xval, imax, 421);
```

まず初めに421をスタックにプッシュして，次に *imax* と *xval* のアドレスをプッシュし，それから *docalc* を呼び出す必要があります。*docalc* から戻ったら，スタックに残っている余計な6バイトをクリアして，計算結果を *ans* と *ans*＋2 に格納します。

コードは次のようになるでしょう。

```
mov     ax,421          ; AX に 421 を置いて
push    ax              ; スタックにプッシュ
push    imax            ; imax をスタックにプッシュ
lea     ax,xval         ; &xval をスタックにプッシュ
push    ax
call    _docalc         ; docalc を呼び出す
add     sp,6            ; スタックをクリア
mov     ans,ax          ; 結果の下位ワード (AX) を ans に
mov     ans+2,dx        ; 上位ワード (DX) を ans + 2 に置く
```

Pascal の引数受け渡しシーケンスを使うとどうなるでしょうか。この場合は，スタックに引数をプッシュする順序が逆になり，制御が戻ってきたときにスタックをきれいにする必要がなくなります。スタックは呼び出されたルーチンがクリアするので，呼び出し側では何もしなくてよいわけです。また，*docalc* の名前はアセンブリソースの中では Pascal の慣例にしたがって書かなければなりません（大文字で下線なし)。

したがって，EXTRN 文は次のようになります。

```
EXTRN   DOCALC:near
```

そして，*docalc* を呼び出すコードは次のようになります。

```
lea     ax, xval        ; &xval をスタックにプッシュ
push    ax
push    imax            ; imax をスタックにプッシュ
mov     ax, 421         ; 421 をスタックにプッシュ
push    ax
call    DOCALC          ; docalc を呼び出す
mov     ans, ax         ; 結果の下位ワード (AX) を ans に
mov     ans+2, dx       ; 上位ワード (DX) を ans + 2 に置く
```

Turbo C と他の言語とのインターフェースをとるために，まず知っておかなければならないことは以上です。

低レベルプログラミング：疑似変数，インラインアセンブリ，割り込み関数

少し低レベルの（システムまわりの）仕事をしたいけれども，独立したアセンブリ言語モジュールを作るわずらわしさは避けたい，というようなことがあるでしょう。Turbo Cは，こんな場合に対しても答えを持っています。疑似変数，インラインアセンブリ，割り込み関数の3つです。この章の残りの部分では，この3つを使って，やりたい仕事をどう処理していくのかをお話します。

疑似変数

システムのCPU(8088/8086/80186/80286プロセッサ)はいくつかのレジスタと特殊な記憶域を持っており，これを使って値を操作します。各レジスタは16ビット（2バイト）の長さを持っており，そのほとんどは特定の用途に用いられますが，汎用に使用できるものもいくつかあります。CPUレジスタの詳細については，この章の最初の節「**メモリモデル**」を参照してください。

低レベルのプログラミングを行なう際には，Cのプログラムからレジスタを直接アクセスしたくなる場合があります。

- システムルーチンを呼び出す前に，レジスタに値をロードしたい。
- レジスタが現在保持している値を知りたい。

Turbo Cでは，**疑似変数**というものが用意されており，これを使うとたいへん簡単にレジスタをアクセスすることができます。疑似変数は，単なる識別名で，各レジスタに対応しています。これは unsigned int または unsigned char の変数のように使うことができます。

表12.6は，使用できる疑似変数の一覧表であり，疑似変数の型，対応するレジスタ名，使用目的も示されています。

表 12.6　Turbo C 疑似変数

疑似変数	型	レジスタ	用途
_AX	unsigned int	AX	汎用/アキュームレータ
_AL	unsigned char	AL	AX の低位バイト
_AH	unsigned char	AH	AX の高位バイト
_BX	unsigned int	BX	汎用/インデクシング
_BL	unsigned char	BL	BX の低位バイト
_BH	unsigned char	BH	BX の高位バイト
_CX	unsigned int	CX	汎用/カウントおよびループ
_CL	unsigned char	CL	CX の低位バイト
_CH	unsigned char	CH	CX の高位バイト
_DX	unsigned int	DX	汎用/データの保持
_DL	unsigned char	DL	DX の低位バイト
_DH	unsigned char	DH	DX の高位バイト
_CS	unsigned int	CS	コードセグメントアドレス
_DS	unsigned int	DS	データセグメントアドレス
_SS	unsigned int	SS	スタックセグメントアドレス
_ES	unsigned int	ES	エキストラセグメントアドレス
_SP	unsigned int	SP	スタックポインタ（SS へのオフセット）
_BP	unsigned int	BP	ベースポインタ（SS へのオフセット）
_DI	unsigned int	DI	レジスタ変数として使われる
_SI	unsigned int	SI	レジスタ変数として使われる

なぜ Turbo C からこれらの変数を直接アクセスしたいのでしょうか。

まず，低レベルのシステムルーチンを呼び出す前にレジスタにある値をセットしたい場合があります。たとえば，INT（割り込み）命令を使ってコンピュータの ROM の中のルーチンを呼び出すことができますが，最初にいくつかのレジスタに必要な値をセットしておかなければなりません。

```
void readchar(unsigned char *ch, unsigned char *attr)
{                                                          /* PC-9801 */
    _AH = 0;                 /* サービスコード：文字と属性の読み込み */
    geninterrupt(0x18);      /* INT 18h を呼び出す */
    *ch = _AL;               /* ASCIIコードを読み込む */
    *attr = _AH;             /* 読み込んだ文字の属性を得る */
}
```

この例では，サービスコードが INT 18h ルーチンに渡されています。また返された値は *ch* と *attr* にコピーされます。

疑似変数は，通常のグローバル変数（型は unsigned int か unsigned char）のように扱ってよいのですが，メモリ上の場所ではなく CPU のレジスタを参照するものなので，制限がいくつかあります。

■疑似変数はアドレスを持っているわけではないので，アドレス演算子(&)をつけることはできません。

■コンパイラは，レジスタを使用するコードを次々に生成するので，疑似変数に値をセットしても，いつまでその値を保持しているかはまったくわかりません。
つまり，*readchar*（上の例）のように，レジスタが使用される直前にセットし，値が得られたらすぐに読み出す必要があるわけです。特に汎用レジスタ(AX，AH，AL など)は，中間的な計算結果を置いておくのに頻繁に使われます。さらに，CPU は汎用レジスタを予期できないように変更することがあります。たとえば，ループの実行やシフト演算の際には CX を，16ビットの乗算の高位ワードの保持には DX を使用します。

■関数呼び出しの前後で，疑似変数が同じ値を保持していると考えることはできません。
たとえば次のコードのような場合，

```
_CX = 18;
myfunc();
i = _CX;
```

関数は，呼び出されたときに，すべてのレジスタを退避するとは限りません。したがって，*i* に18が代入されるかどうかはわからないのです。関数呼び出しの前後で同じ値を持っているとみなすことができるのは，_CS,_BP,_SI,_DI だけです。

■いくつかのレジスタは，変更すると予期できない結果をもたらすことがあります。たとえば，_CS,_SS,_SP,_BP に値を直接代入すると，プログラムはほとんどの場合エラーを引き起こします。コンパイラが出力する機械語コードは，これらのレジスタを頻繁に使用するからです。

インラインアセンブリ言語を使う

アセンブリ言語ルーチンを別個に作って，Turbo C プログラムとリンクする方法についてはすでに述べましたが，Turbo C では，C のプログラムの中にアセンブリ言語コードを書くこともできます。これは**インラインアセンブリ**と呼ばれています。

C プログラムの中でインラインアセンブリを使うには，-B コンパイラオプションを指定します。指定しない場合，コンパイラはインラインアセンブリに出会うと警告を出力し，-B オプションでコンパイルをリスタートします。# pragma inline 文をソース中に入れておくと，これを避けることができます。コンパイラが# pragma inline 文を見つけると，そこで-B オプションをセットするからです。

この機能を使う場合は，Turbo アセンブラ（TASM）が必要です。コンパイラはまずアセンブリファイルを作成し，その後 TASM を呼び出して.OBJ ファイルを作成します。

もちろん，8086の命令セットとアーキテクチャに習熟していなければなりません。完全なアセンブリ言語ルーチンを書くのではありませんが，使用する命令がどのように動くか，命令をどう使えばよいかについては知っている必要があります。

インラインアセンブリ命令であることを示すためには，予約語 asm を次のように命令の先頭につけます。

```
asm    命令コード  オペランド  ;（または改行）
```

■**命令**コードは，8086の正しい命令でなければなりません（使用できる命令コードの一覧表は後で示します)。

■**オペランド**にはその命令コードに対して許されるオペランドを指定し，C の定数，変数，ラベルを参照することができます。

■命令の最後にはセミコロン（;）置くか，改行を行ないます。どちらも asm 文の終わりを意味します。

複数の asm 文を，セミコロンで区切って同じ行に書くことはできますが，1つの asm 文を複数行にわたって書くことはできません。

（TASM のように）セミコロンでコメントを開始することはできません。コメントは次

のようにCの書き方を使います。

```
asm mov ax,ds;                          /* このコメントは OK */
asm pop ax; asm pop ds; asm iret;       /* これも正しい */
asm push ds                             ; このコメントは正しくない!!
```

最後の行はエラーを起こすことに注意してください。コメントが正しくないからです。

'**命令コード オペランド**'は，Turbo CがCのソースコードから生成するアセンブリ言語コードの中に，そのまま埋め込まれていきます。Cのシンボルは，適当なアセンブリ言語の対応するものに置きかえられます。

インラインアセンブリ機能は完全なアセンブラではないので，多くのエラーは検出されません。TASMではすべてのエラーが検出されますが，オリジナルのCの行番号が失われているので，エラーの起こった箇所はわからないことになります。

各asm文はCの文1つとして数えられます。例を次に示します。

```
myfunc()
{
    int  i;
    int  x;

    if  (i > 0)
        asm  mov  x,4
    else
        i = 7;
}
```

これはTurbo Cの正しいif文です。mov x,4の後ろにセミコロンを置く必要はありません。asm文は，Cの文の中で唯一改行の有無に依存する文であり，他のCの文とは異なるわけですが，いくつかのUNIX上のコンパイラで採用されている機能です。

アセンブリ文は関数の中で実行文として使用することも，関数の外側で外部宣言として使用することもできます。関数の外側に置かれたアセンブリ文はDATAセグメントに，関数の内側に置かれたアセンブリ文はCODEセグメントに配置されます。

次に，（この章の「**値の返し方**」で示した）関数*min*のインラインアセンブリ版を示します。

```
int min (int V1, int V2)
{
    asm  mov  ax,V1
    asm  cmp  ax,V2
    asm  jle  minexit
    asm  mov  ax,V2
    minexit:
    return (_AX);
}
```

この例を見ると，Turbo Cのインラインアセンブリが.ASMルーチンより融通のきく，より強力なものであるかがわかります。このインラインアセンブリの関数は，ラージコード，スモールコードのいずれのモデルに対しても，またPascalの呼び出し慣例，Cの呼び出し慣例のいずれに対しても使うことができるからです。

.ASMモジュールを作る場合は，使用するメモリモデルや呼び出し慣例（CかPascalか）によって変更が必要でした。また，*min*の.ASM版では，いつも引数のオフセットや識別名のスペル（ *min*か*MIN*か）を気にしなければなりませんでした。このインラインアセンブリ版ではその必要はないのです。

注意：バージョン2.0で新たに追加された__**emit**__という機能を使うと，Turbo Pascal形式のインラインコードを生成することができます。__**emit**__の詳しい情報については，リファレンスガイド第2章を参照してください。

8086の命令コードはすべて，インラインアセンブリ文に含めることができます。Turbo Cコンパイラが受け入れる命令は次の4つに分類することができます。

■通常の命令	8086の命令コードセット
■文字列命令	特別な文字処理コード
■ジャンプ命令	さまざまなジャンプ命令コード
■アセンブリ指令	データの割当てと定義

コンパイラは，たとえオペランドが間違っていて，アセンブラであればエラーが出力されるような場合でも，まったくエラーを検出しません。オペランドの厳密な書式はコンパイラによっては規定されないのです。

命令コード

表12.7に，通常の命令として使用できる命令のニーモニックの一覧表を示しました。

表12.7　命令コードのニーモニック

aaa	fcom	fldl2t	fsub	or
aad	fcomp	fldlg2	fsubp	out
aam	fcompp	fldln2	fsubr	pop
aas	fdecstp**	fldpi	fsubrp	popa
adc	fdisi	fldz	ftst	popf
add	fdiv	fmul	fwait	push
and	fdivp	fmulp	fxam	pusha
bound	fdivr	fnclex	fxch	pushf
call	fdivrp	fndisi	fxtract	rcl
cbw	feni	fneni	fyl2x	rcr
clc	ffree**	fninit	fyl2xp1	ret
cld	fiadd	fnop	hlt	rol
cli	ficom	fnsave	idiv	ror
cmc	ficomp	fnstcw	imul	sahf
cmp	fidiv	fnstenv	in	sal
cwd	fidivr	fnstsw	inc	sar
daa	fild	fpatan	int	sbb
das	fimul	fprem	into	shl
dec	fincstp**	fptan	iret	shr
div	finit	frndint	lahf	stc
enter	fist	frstor	lds	std
f2xm1	fistp	fsave	lea	sti
fabs	fisub	fscale	leave	sub
fadd	fisubr	fsqrt	les	test
faddp	fld	fst	mov	wait
fbld	fld1	fstcw	mul	xchg
fbstp	fldcw	fstenv	neg	xlat
fchs	fldenv	fstp	not	xor
fclex	fldl2e	fstsw		

注意：80186命令のニーモニックをインラインアセンブリ文の中で使う場合は，-1コンパイラオプション（**O/C/C/**Instruction set..80186/80286）を指定しなくてはなりません。こうすると，Turbo アセンブラがニーモニックを受け入れる適当な文をアセンブリ言語コードに出力することになります。もし古いアセンブラを使っている場合には，これらのニーモニックはまったくサポートされません。

注意をもう1つ：浮動小数点エミュレーションを使う（-f オプション）ルーチンの中でインラインアセンブリを使用する場合は，＊＊のついた命令コードはサポートされません。

文字列命令

命令コードに加えて，次のような文字列命令が単独で，あるいは繰り返しプレフィクス（接頭辞）とともに使用することができます。

表 12.8　文字列命令

cmps	insw	movsb	outsb	scasw
cmpsb	lods	movsw	outsw	stos
cmpsw	lodsb	msb	scas	stosb
ins	lodsw	outs	scasb	stosw
insb	mov			

繰り返しプレフィクス

次のような繰り返しプレフィクスが使用できます。

rep　**repe**　**repne**　**repnz**　**repz**

ジャンプ命令

ジャンプ命令は特別に扱われます。命令それ自身にラベルを含めることはできないので，Cのラベル（「**ジャンプ命令とラベルの使い方**」で説明します）へジャンプすることになります。使用できるジャンプ命令は以下のとおりです。

表 12.9　ジャンプ命令

ja	jge	jnc	jnp	js
jae	jl	jne	jns	jz
jb	jle	jng	jnz	loop
jbe	jmp	jnge	jo	loope
jc	jna	jnl	jp	loopne
jcxz	jnae	jnle	jpe	loopnz
je	jnb	jno	jpo	loopz
jg	jnbe			

アセンブリ指令

Turbo C インラインアセンブリでは次のようなアセンブリ指令が使用できます。

```
db      dd      dw      extrn
```

インラインアセンブリによるデータと関数の参照

asm 文では C のシンボルを使うことができます。Turbo C は，C のシンボルをアセンブリ言語のオペランドに変換し，識別名の前に下線をつけます。自動（ローカル）変数，レジスタ変数，関数の引数も含め，すべてのシンボルを使用することができます。

一般に C のシンボルは，アドレスオペランドが置ける所であればどこにでも指定することができます。もちろんレジスタ変数は，レジスタが置ける所ならどこにでも指定できます。

アセンブラは，インラインアセンブリ命令のオペランドを解析しているときに識別名に出会うと，C のシンボルテーブルの中からその識別名を探します。ただし，8086のレジスタ名はこの探索からは除外されます。レジスタ名は大文字小文字どちらで書いてもかまいません。

インラインアセンブリとレジスタ変数

関数の中の register 宣言のうち，最も頻繁に使用される2つがレジスタ変数として扱われ，他は自動（ローカル）変数として扱われます。レジスタに割り当てることができない変数に対する予約語 register は無視されます。

レジスタに割り当てることができるのは，short 型と int 型（およびその unsigned 型）の変数，または2バイトのポインタ変数だけです。レジスタ変数には SI と DI が使われます。関数の中でレジスタ宣言をしていなければ，SI と DI は中間書き込みレジスタとしてインラインアセンブリコード内で自由に使用してかまいません。C の関数に入るときに，呼び出し側の SI と DI の値は自動的に退避および回復が行なわれます。

レジスタ宣言が1個ある場合は，インラインアセンブリでは，SI あるいは DI によってレジスタ変数の値を使う（または変更する）ことはできますが，内部的なレジスタ変数の扱いに影響を与えないようにするためには，C のシンボルを使った方がよいでしょう。

インラインアセンブリ，オフセット，サイズオーバーライド

プログラミングの際に，ローカル変数の正確なオフセットを知っている必要はありません。名前を使用するだけで，正しいオフセットが得られます。

しかし，アセンブリ命令に対して，適当なWORD PTRやBYTE PTR，あるいは他のサイズオーバーライドを必要とする場合もあります。DWORD PTRオーバーライドは，LESや間接的なfarコール命令に対して必要になります。

Cの構造体のメンバを使う

インラインアセンブリ文の中でも，構造体のメンバを通常どおり，つまり変数名.メンバ名で参照することができます。これは変数を扱うのと同じで，値を代入したり，読み出したりすることができます。また，メンバ名を数値定数として直接（変数名なしで）参照することもできます。この場合の定数は，そのメンバの構造体の先頭からのオフセット（バイト単位）に等しくなります。次のプログラムを考えてみましょう。

```
struct  myStruct {
    int      a_a;
    int      a_b;
    int      a_c;
} myA ;

myfunc()
{
    ...
    asm  mov  ax, myA.a_b
    asm  mov  bx, [di].a_c
    ...
}
```

構造体型 *myStruct* は3個のメンバ（*a_a*, *a_b*, *a_c*）を持ち，*myA* は *myStruct* 型の変数であると宣言しています。最初の asm 文は，*myA.a_b* の値を AX レジスタにセットします。2番目は，アドレス[di]＋オフセット（*a_c*）に存在する値を BX レジスタにセットします（DI に入っているアドレスを取り出し，*a_c* の *myStruct* の先頭からのオフセットを加える）。2つの asm 文から，次のようなコードが生成されます。

```
mov   ax, DGROUP : myA+2
mov   bx, [di+4]
```

どのような場合にこの機能が必要になるのでしょうか。レジスタ（たとえばDI）に *myStruct* 型の構造体のアドレスをロードしていれば，メンバ名を使って直接メンバを参照することができます。メンバ名は，数値定数が置けるアセンブリ文オペランドならばどこにでも指定することができます。

構造体のメンバは，通常のCのシンボルではなく，メンバ名であることを示すためにピリオド(.)をその前につけなければなりません。メンバ名は，コンパイラの出力のアセンブリソースでは，構造体メンバの数値オフセット(この例では，*a_c* の数値オフセットは4)で置きかえられます。ただし型に関する情報は失われます。このようにして，メンバ名はアセンブリ文においてコンパイル時定数として使うことができます。

しかし，1つだけ制約があります。インラインアセンブリで使用する2つの構造体が同じ名前のメンバを持っている場合，ピリオドとメンバ名の間に（キャストのように）カッコで囲んだ構造体名をはさんでおかなければなりません。たとえば次のようにします。

```
asm  mov  bx,[di].(struct tm)tm_hour
```

ジャンプ命令とラベルを使う

インラインアセンブリでは，条件つきジャンプ，無条件ジャンプ，ループ命令が使用できます。ただし，これらは関数の中でのみ有効です。asm 文ではラベルを指定できないので，ジャンプ命令の行き先にはCの goto ラベルを使わなければなりません。直接 far ジャンプは使えません。

間接ジャンプは使用できます。間接ジャンプを使用するときには，ジャンプ命令のオペランドとしてレジスタ名を指定します。次に示すコードでは，ジャンプはCの goto ラベル a に飛びます。

```
int  x()
{
a:                          /* gotoラベル a:     */
     ...
     asm  jmp  a            /* a: にジャンプする */
     ...
}
```

割り込み関数

8086では，先頭の1024バイトのメモリは予約されており，256組の far ポインタ（割り込みベクター）が置かれています。それぞれの far ポインタは**割り込みハンドラ**と呼ばれる特別なシステムルーチンを指しています。このルーチンは，次の8086命令で呼び出すことができます。

```
int   割り込み番号
```

割り込み番号は00h～FFh の間の値をとります。この命令が実行されると，コンピュータはコードセグメント（CS），命令ポインタ（IP），ステータスフラグを退避し，割り込みを禁止し，割り込み番号に対応する割り込みベクタが指している場所へ far ジャンプを行ないます。たとえば，よく出会う割り込みに次のものがあります。

```
int   21h
```

これは多くの MS-DOS ルーチンを呼び出すものです。しかし，使用されていない割り込みベクタもかなりあるので，独自の割り込みハンドラを書いて，それを指す far ポインタを未使用のベクターに置くことができます。

Turbo C で割り込みハンドラを書く場合は，その関数を interrupt 型と定義しなければなりません。たとえば次のようになります。

```
void  interrupt  myhandler( bp, di, si, ds, es, dx,
                            cx, bx, ax, ip, cs, flags, ... );
```

すべてのレジスタを引数として渡しているので，ハンドラ内では疑似変数を使わずにレジスタを使用したり，変更したりすることができます。また，このハンドラにはさらに引数（flags,...）を渡せることに注意してください（これは正しく定義しなければなりませんが）。

interrupt 型の関数では，自動的に（SI,DI,BP に加えて）AX～DX，ES および DS レジスタが退避されます。以上のレジスタは，割り込みハンドラ終了時には回復されます。

割り込みハンドラでは，どんなメモリモデルの場合でも，浮動小数点演算を行なうことができます。ただし，8087/80287を使用するハンドラは，コプロセッサの状態を退避し，終了時にはそれを回復しなければなりません。

割り込み関数は，引数に変更を加えることができます。宣言された引数を変更すると，割り込みハンドラから戻るときの，対応するレジスタの値を変更することになります。こ

れは，MS-DOSのINT 21hのような，ユーザサービスとして動作する割り込みハンドラを使用する場合に便利です。なお，割り込み関数はIRET (Interrupt Return) 命令で終わることに注意してください。

さて，どんな場合に独自の割り込みハンドラが必要になるのでしょうか。まず，ほとんどのメモリ常駐ルーチンは割り込みハンドラとして動作します。こうしたルーチンは，自分自身を割り込みハンドラとしてインストール（システムに組み込む）します。そして，何か特別な動作（特定のキーが押されるなど）あるいは周期的な動作（クロックの経過時間など）が起こったときに割って入り，どのような動作を行なうべきかを調べてそれを実行します。必要な動作が終了したら，再び元のルーチンに制御を戻します。

低レベルのプログラミングの実際

例をいくつか使って低レベルプログラミングのやりかたを何通りか見てきました。もう少しお話していきましょう。最初は，害がなくなおかつ効果が“見える”（ここでは聞こえる）割り込みハンドラを実際に書いてみることにします。このハンドラは呼び出されるとシステムのブザーを鳴らします。

まず，関数そのものを書かなければなりません。次のようなものです。

```
#include <dos.h>

/* この関数は PC-9801 シリーズでのみ動作します. */

void  interrupt  mybuzzer( unsigned bp, unsigned di, unsigned si,
                           unsigned ds, unsigned es, unsigned dx,
                           unsigned cx, unsigned bx, unsigned ax)
{
    int  i, j;
    unsigned char bcount = ax >> 8;

    for (i = 0; i <= bcount; i++) {

        outportb(0x37, 0x06);        /* ブザーを鳴らす */
        for (j = 0; j <= 100; j++)
            ;                        /* 空の文 */

        outportb(0x37, 0x07);        /* ブザーを止める */
        for (j = 0; j <= 100; j++)
            ;                        /* もう1つ空の文 */
    }
}
```

次に，この割り込みハンドラをインストールするための関数が必要となります。その関数には，ハンドラ関数のアドレスと割り込み番号 (0～255または0x00～0xFF) を与えるこ

とになります。関数は次の3つのことを行ないます。

- ■割り込みを禁止して，ベクターテーブルの書き換え中に変なことが起きないようにする。
- ■ハンドラ関数のアドレスを，適当な記憶場所に格納する。
- ■割り込みを可能にして，すべてが元どおり動くようにする。

インストールルーチンは次のようになります。

```
void  install(void interrupt (*faddr)(), int inum)
{
    setvect(inum, faddr);
}
```

最後に，このブザールーチンをテストする関数を書きます。

```
void  testbuzzer(unsigned char bcount, int inum)
{
    _AH = bcount;
    geninterrupt(inum);
}
```

全部書き終えたら，次のような **main** 関数を作ります。

```
main()
{
    char ch;

    install(mybuzzer, 10);
    testbuzzer(100, 10);
    ch = getch();
}
```

浮動小数点ライブラリの使い方

Cには，数値の型が2つあります。整数（int, short, long, 他）と浮動小数点数（float, double, long double）です。コンピュータのプロセッサは，整数は容易に扱えるようになっていますが，浮動小数点数を扱うにはかなりの時間と努力を必要とします。

しかし，iAPX86ファミリィのプロセッサには，数値演算コプロセッサとして8087および80287が用意されています。

8087および80287（ここではどちらも8087またはコプロセッサと呼んでいます）はコンピュータに組み込むことができる特別な数値プロセッサで，浮動小数点命令を非常に高速に実行します。浮動小数点の演算をたくさん必要とする場合は，このコプロセッサが欲しくなるでしょう。CPU は，特殊な割り込みによって8087とのインターフェースをとります。

Turbo C は，コンピュータの機能を引き出し，ユーザの要求を満足させるプログラムが作成できるように設計されています。

- 浮動小数点演算をまったく行なわない場合は，コンパイラにそのことを知らせることができます。
- 浮動小数点演算は行なうけれども8087は装着されていない場合は，Turbo C に，8087が入っているかのように思わせる特別なルーチンをリンクするように指示することができます。こうして作成したプログラムは，コプロセッサが入っているシステムで実行すると自動的に8087チップが使用され，プログラムは高速に実行されます。
- コプロセッサを持っているシステムでのみ走るプログラムの場合は，コンパイラに，常に8087を使用するコードを生成するように指示することができます。

以下の TCC および TLINK の例では，-L と -I に正しいパスが指定されているファイル TURBOC.CFG があり，また，ライブラリファイルとスタートアップファイルはサブディレクトリ ¥LIB に置かれているものとします。

8087/80287 チップのエミュレーション

浮動小数点演算は行なうけれども，数値演算コプロセッサはコンピュータに入っていない場合はどうすればよいでしょうか。また，プログラムを走らせるコンピュータがコプロセッサを持っているかどうかわからない場合はどうでしょうか。心配はいりません。どち

らの場合も，Turbo C はうまく対処できます。

エミュレーションのオプションを指定すると，コンパイラは，8087が存在するかのようなコードを生成し，エミュレーションライブラリ (EMU.LIB) をリンクします。プログラムを実行すると，8087があればそれを使い，なければ8087を**エミュレート**する特別なソフトウェアを使います。

エミュレーションライブラリは次のような働きをします。

- ■プログラムの実行が始まると，C のスタートアップコードが，8087が存在するかどうかを決める。
- ■コプロセッサがある場合，プログラムは，8087に対する特別な割り込みを直接8087チップに渡すことにする。
- ■コプロセッサがない場合，プログラムは，8087への割り込みがあったら，それをエミュレーションルーチンに渡すようにする。

第6章で作った RATIO.C を次のように書き変えてみましょう。

```
main()
{
    float  a,b,ratio;

    printf("Enter two values:  ");
    scanf("%f %f",&a,&b);
    ratio = a/b;
    print("The ratio is %0.2f¥n",ratio);
}
```

TC（統合環境）を使用している場合は，まず **Options** メニューに入って，**Compiler** を選択し，**Code generation** を選択してから **Floating point** の項目を **Emulation** にします。プログラムをコンパイルすると，Turbo C は自動的に適切なオプションとライブラリィを選択します。

TCC（スタンドアロンコンパイラ）を使用する場合は，次のように指定します。

```
tcc -mX ratio
```

RATIO.C はそれだけで独立したプログラムなので，TCC は自動的に TLINK を呼び出し，必要なスタートアップファイルとライブラリファイルをリンクして，RATIO.EXE を作成します。

TCC のコマンドラインで，-c オプション（コンパイルのみ）を指定した場合は，後で自分でリンクを行なわなくてはなりません。この場合，適当な数学関数ライブラリィファイル (MATH*x*.LIB) と EMU.LIB ファイルを指定してください。エミュレーションオプシ

ョン（-f）はオンがデフォルトなので，TURBOC.CFG ファイルの中に他の浮動小数点オプション（-f-または-f87）の指定がないのであれば必要ありません。
TLINK のコマンドは次のようになります。

```
tlink lib¥c0X ratio, ratio, ratio, lib¥emu.lib lib¥mathX.lib lib¥cX.lib
```

X はメモリモデルを示す英字です（t＝タイニィ，s＝スモール，c＝コンパクト，m＝ミディアム，l＝ラージ，h＝ヒュージ）。これで，C0*x*.OBJ と RATIO.OBJ がリンクされ，EMU.LIB と MATH*x*.LIB および C*x*.LIB ライブラリが取り込まれて，RATIO.EXE と RATIO.MAP が作成されます。

注意：TLINK のコマンドは1行で与えなければなりません。また，指定するライブラリの順番はたいへん重要ですので注意してください。

8087/80287 演算用コプロセッサチップを使う

8087または80287チップが装備したシステムで実行すると決まっている場合は，その利点を生かしてプログラムを作成することができます。この場合は，エミュレーションルーチン（EMU.LIB）をリンクしなくてよいので，EXE ファイルが小さくなります。
TC（統合環境）を使用している場合は，**O/C/C/F**loating point を8087/80287に設定します。プログラムをコンパイルしてリンクすると，適当なオプションとライブラリが自動的に選択されます。
TCC（スタンドアロンコンパイラ）の場合は，次のように-f87オプションを指定する必要があります。

```
tcc -f87 -mx ratio
```

これは，コンパイラに，8087/80287のインラインコードを生成するように指示しています。TLINK が呼び出されると，FP87.LIB と MATH*x*.LIB がリンクされます。
自分でリンクを行なう場合は，次のように適当な数学ライブラリ（メモリモデルによる）と FP87ライブラリを指定しなければなりません。

```
tlink lib¥c0X ratio, ratio, ratio, lib¥fp87.lib lib¥mathX.lib lib¥cX.lib
```

X は，必要なメモリモデルを示す英字です。

浮動小数点演算を行なわない場合

プログラムに浮動小数点演算がまったくない場合は，リンカは浮動小数点ライブラリ（MATH*x*.LIB，EMU.LIB，FP87.LIB）をいっさいリンクしません。コマンドラインでそのライブラリを指定しても，リンクされません。リンカのコマンドラインで，こうしたライブラリを指定しないようにすると（プログラムが浮動小数点演算を行なわない場合は）リンクの時間を短縮することができます。

次のプログラム（RATIO.C）をコンパイル・リンクするとしましょう。

```
main()
{
     int  a,b,ratio;

     printf("Enter two values:  ");
     scanf("%d %d",&a,&b);
     ratio = a/b;
     printf("The ratio is %d¥n",ratio);
}
```

このプログラムは浮動小数点ルーチンを使用していないので，エミュレーションを指定しても，浮動小数点演算なしを指定してもかまいません。

TCの場合は，**O/C/C/Floating point**をEmulationに設定しておきます（Emulationがデフォルトです）。リンクのステップでは浮動小数点ライブラリも読み込まれますが，実際には何もリンクされません。

リンクの時間を短縮したい場合は，**Floating point**トグルをNoneにセットしてください。Noneを指定してこのプログラムをコンパイル・リンクすると，Turbo Cはどんな浮動小数点ライブラリ・数学関数ライブラリもリンクしようとしません。

TCCの場合は，次のように-f-オプションを指定しなくてはなりません。

```
tcc -f- -mX ratio.c
```

これは，コンパイラに，プログラムの中で浮動小数点演算を使っていないことを伝えます。

自分でリンクを行なう場合，数学関数ライブラリを指定する必要はありません。TLINKコマンドは次のようになります。

```
tlink lib¥c0x ratio, ratio, ratio, lib¥cx.lib
```

環境変数 87

8087エミュレーション（**O/C/C/Floating point..Emulation** または-f-オプション）でプログラムを作成した場合，スタートアップモジュール（C0*x*.OBJ）は，プログラムの実行を開始するときに，8087自動検出ルーチンを使用して，8087が使用できるかどうかをチェックします。

8087が装備されていれば，プログラムはそれを利用し，なければエミュレーションルーチンを使用します。

しかし，このデフォルトの自動検出をやめたい場合もあります。たとえば，自分のシステムは8087を持っているけれども，プログラムを実際に走らせるシステムは持っていないような場合に，対象のシステムで正しく走るかどうかを調べたいということがあるでしょう。

Turbo C では，このような場合のためにスタートアップコードの自動検出機能を無効にすることもできます。環境変数87がそれです。

環境変数87は，MS-DOS の SET コマンドを使って，次のように設定します。

```
A>SET 87=N
```

または

```
A>SET 87=Y
```

環境変数87を N（No）とすると，スタートアップコードに（システムに8087が入っていても）8087を使わないと知らせることになります。

逆に環境変数87を Y（Yes）とすると，コプロセッサが存在し，それを使用することを意味します。87＝Y と設定しているにもかかわらず，実際には8087がシステムにない場合には，プログラムは暴走してしまうので注意してください。

環境変数87は，デフォルトの自動検出機能に優先します。したがってプログラムの実行が始まると，スタートアップコードはまず環境変数87が定義されているかどうかをチェックします。

■変数87が定義されている場合，スタートアップコードは自動検出を行なわず，プログラムはあらかじめセットされたモードで実行される。

■変数87が定義されていない場合，スタートアップコードは自動検出を行なって，プログラムはその結果にしたがって実行される。

環境変数87がすでに設定されている場合に，これを未定義にしたいときは，次のようなMS-DOS コマンドを入力します。

```
A>SET 87=
```

等号を入力した後，単に**リターン**キーを押します。

レジスタと 8087

浮動小数点演算を行なう場合に，レジスタに関して知っておく必要のある点がいくつかあります。

まず第一に，8087エミュレーションモードでは，レジスタのラップアラウンドはサポートされません。

第二は，インラインアセンブリと浮動小数点演算を一緒に使う場合は，レジスタを使用する際には特別に注意が必要です。これは，Turbo C が関数を呼ぶ前に，8087のレジスタセットは空にされるからです。フリーのレジスタが充分あるという確信がなければ，コプロセッサを使用する関数を呼び出す前に，8087レジスタをポップしてセーブしておく必要があるでしょう。

浮動小数点演算とともに matherr を使う

プログラムの実行中に，浮動小数点ルーチンでエラーが検出されると，そのルーチンは自動的に **_matherr** を呼び出し，引数をいくつか渡します。**_matherr** は，その引数の値で **exception** 構造体(MATH.H の中で定義されています)を埋めた後，その構造体を指すポインタを引数として **matherr** を呼び出します。

matherr は，ユーザ独自のエラー処理ルーチンを書くためのフックとして使うことができます。**matherr** は，デフォルトでは何もせず0を返しますが，**matherr** を修正して，浮動小数点ルーチンのエラーを望むように操作することができます。そのように修正された **matherr** は，エラーが解決された場合はゼロ以外の値を，解決されなかった場合は0を返すことになっています。

matherr と **_matherr** の詳細については，リファレンスガイドの第2章を参照してください。

注意とヒント

Turbo C に必要な RAM の大きさ

Turbo C は，コンパイル中に，中間的なデータはディスクに出力しません（ディスクに出力されるのは.OBJ ファイルだけです)。パスの間の中間的なデータは RAM に置かれます。このため，コンパイラが使用できるメモリの量がたりないときに，Out of memory ... というメッセージが出力される場合があります。

この問題を解決するためには，関数を小さくしたり，大きな関数を含むファイルを分割したりします。また，Turbo C が使用できるメモリを増やすために，RAM に常駐するプログラムを削除するのも1つの手です。

Pascal の慣例を使うべきか

Pascal の慣例は，少なくともこの章をよく読んでよく理解するまでは，使わない方がよいでしょう。

もし，メインファイルを Pascal の呼び出し慣例でコンパイルしたい場合は，**main** を C の関数（cdecl）として宣言することを忘れてはなりません。

```
cdecl  main(int argc, char *argv[], char *envp[])
```

まとめ

この章では，Turbo C における3種類の低レベルプログラミング(疑似変数，インラインアセンブリ，割り込み関数）について述べてきました。アセンブリ言語も含めて他の言語とのインターフェースをどのように行なうかについてもみてきました。浮動小数点ルーチンを使う場合の細かい点についても検討してきました。そして，8086における6つのメモリモデルがどのような関連をもつかについて理解ができました。さあ，こうしたテクニックを使って，コンピュータを自由にコントロールしていくのは，あなたです。幸運を祈ります。

付録 A

Turbo C エディタ

紹介

Turbo C に組み込まれているエディタは，主にソースプログラムを書くのに使われることを想定して作られました。Turbo Pascal や SideKick のエディタ，MicroPro の WordStar を使ったことがあれば，Turbo C エディタはすぐに使えるはずです。コマンドはほとんど同じだからです。この付録の最後に，Turbo C エディタと WordStar のコマンドの相違を示しておきます。Turbo C エディタは WordStar と違って，アンドゥ機能を持っており，他の行に移る前であれば，その行に施した変更を取り消すことができます。このコマンド（***CTRL-Q L***）は「**その他の編集コマンド**」の中で述べられています。

TC エディタは，3.2(あるいはそれ以上)の LIM/EMS(Lotus/Microsoft/Intel Expanded Memory Specification）に対応する EMM（Extended Memory Manager）ドライバが組み込まれたシステム上では拡張メモリをサポートします。Turbo C は，起動時に EMS メモリが使用可能かどうかを決定します。EMS が使用可能であれば，自動的にエディタバッファを拡張メモリに置きます。これによって，プログラムのコンパイル・実行のための RAM メモリを64K 増やすことができます。EMS が使用可能であっても Turbo C で使いたくない場合は，TCINST の **O**ptions/**E**nvironment/**O**ptions for editor メニューで，バッファのタイプをメインメモリにセットしてください(IBM PC の場合は，EMS の使用をオフにセットしてください)。

また，PC-9801では，TCINST の **M**ode for display で，テキスト画面のみを使用するように設定している場合には，グラフィック VRAM のバンク#1（裏 VRAM）をエディタバッファとして使うこともできます。これには，TCINST の **O**/**E**/**O**ptions for editor のバッファのタイプを，**G**VRAM bank #1にセットしてください。

エディタバッファの設定の詳細については付録 D を参照してください。

エディタに入る，エディタから出る

エディタを呼び出すには，Turbo C のメインメニューから Edit を選びます。Edit ウィンドウが"アクティブ"ウィンドウになり，Edit ウィンドウのタイトルがハイライト表示され，タイプした文字が入力される位置を示すカーソルが Edit ウィンドウに現われます。

テキストを入力するには，タイプライタを打つように単にキーをたたくだけです。行の最後には**リターン**キーを押します。

エディタから出てメインメニューに戻るには，***F10*** を押します（Edit ウィンドウの中のテキストはスクリーンに表示されたままです）。

Edit ウィンドウのステータス行

Edit ウィンドウの一番上の行（バーと呼びます）にステータス行があります。これは，現在編集中のファイルに関する情報，"カーソルはどこにあるか"，"どのような編集モードになっているか"などを示すものです。

```
Line    Col    Insert Indent Tab Fill Unindent * X:FILENAME.EXT
```

Line カーソルが何行目にあるかを示します。

Col カーソルが何列目にあるかを示します。

Insert これはエディタが挿入（インサート）モードになっていることを示します。このモードでは，キーボードから入力した文字はカーソルの位置に挿入されていき，カーソル以降にあるテキストは右へ移動します。***INS*** キーまたは ***CTRL-V*** で，挿入(Insert)モードと上書き(Overwrite)モードを切り換えることができます。
上書きモードでは，キーボードから入力された文字は，現在カーソルがある文字に上書きされます。

Indent 自動的にインデント(段下げ)が行なわれることを意味します。***CTRL-O I*** によって自動インデントのオン/オフが切り換わります。

Tab タブが使えるかどうかを示します。***CTRL-O T*** によってタブモードのオン/オフが切り換わります。

Fill	最適フィルがオンになっていることを示しています。タブモードがオンのときに最適フィルモードもオンになっていると，各行の先頭の空白部分は，タブ文字と空白文字の組み合わせで最適に（最もバイト数が少なくなるように）埋められます。 最適フィルモードは，***CTRL-O F*** でオン/オフが切り換わります。
Unindent	逆インデントがオンになっていることを示しています。逆インデントがオンの場合，行の最初の非空白文字にカーソルがあるときに ***BS*** キー（または ***CTRL-H***）を押すと，インデントのレベルが1つ戻ります。 逆インデントモードは，***CTRL-O U*** でオン/オフが切り換わります。
*****	ファイル名の前につけられるアスタリスクは，そのファイルは変更されており，またセーブされていないことを示します。
X:FILENAME.EXT	現在編集中のファイルの，ドライブ名(X:)，ファイル名（FILENAME），拡張子（.EXT）を示します。NONAME.C と表示されている場合は，まだファイル名を指定していないことになります（NONAME.C は，Turbo C のデフォルトのファイル名です）。

エディタのコマンド

Turbo C エディタは約50個のコマンドを持っており，カーソルを移動する，画面単位で移動する，文字列を探索したり置換したりする，などの操作を行なうことができます。こうしたコマンドは，次の5つのカテゴリに大別することができます。

■基本カーソル移動コマンド
■クイックカーソル移動コマンド
■挿入および削除コマンド
■ブロックコマンド
■その他のコマンド

表 A.1 にコマンドの要約を示しました。左側にコマンドの定義，右側にコマンドを呼び出すための通常のキーを示しています。この表の後で，各編集コマンドについて説明していきます。

表 A.1　エディタコマンドの要約

基本カーソル移動コマンド：

1文字左へ	***CTRL-S*** または ←（左矢印）
1文字右へ	***CTRL-D*** または →（右矢印）
1単語左へ	***CTRL-A***
1単語右へ	***CTRL-F***
1行上へ	***CTRL-E*** または ↑（上矢印）
1行下へ	***CTRL-X*** または ↓（下矢印）
スクロールアップ	***CTRL-W***
スクロールダウン	***CTRL-Z***
1画面アップ	***CTRL-R*** または ***ROLLDOWN***（***PgUp***）
1画面ダウン	***CTRL-C*** または ***ROLLUP***（***PgDn***）

クイックカーソル移動コマンド：

行の初めへ	***CTRL-Q S*** または ***HOME***（***Home***）
行の終わりへ	***CTRL-Q D*** または ***SHIFT-HOME***（***End***）

ウィンドウの最初へ	***CTRL-Q E***
ウィンドウの最後へ	***CTRL-Q X***
ファイルの最初へ	***CTRL-Q R*** または ***CTRL-ROLLDOWN***（***Ctrl-PgUp***）
ファイルの最後へ	***CTRL-Q C*** または ***CTRL-ROLLUP***（***Ctrl-PgDn***）
ブロックの最初へ	***CTRL-Q B***
ブロックの最後へ	***CTRL-Q K***
直前のカーソルの位置へ	***CTRL-Q P***

挿入および削除コマンド：

挿入モードのオン/オフ	***CTRL-V*** または ***INS***（***Insert***）
行の挿入	***CTRL-N***
行の削除	***CTRL-Y***
行の終わりまで削除	***CTRL-Q Y***
カーソルの左の文字の削除	***CTRL-H*** または ***BS***（***Backspace***）
カーソル位置の文字の削除	***CTRL-G*** または ***DEL***（***Delete***）
カーソルの右の単語の削除	***CTRL-T***

ブロックコマンド：

ブロックの最初のマーク	***CTRL-K B***
ブロックの最後のマーク	***CTRL-K K***
1語をマーク	***CTRL-K T***
ブロックの複写	***CTRL-K C***
ブロックの削除	***CTRL-K Y***
ブロックの表示のオン/オフ	***CTRL-K H***
ブロックの移動	***CTRL-K V***
ディスクからのブロックの読み込み	***CTRL-K R***
ディスクへのブロックの書き出し	***CTRL-K W***
プリンタへのブロックの印字	***CTRL-K P***
ブロックのインデント	***CTRL-K I***
ブロックの逆インデント	***CTRL-K U***

その他のコマンド：

タブ	***CTRL-I*** または ***TAB***（***Tab***）
制御文字の入力	***CTRL-P***
コマンドの中断	***CTRL-U***

自動インデントのオン/オフ	***CTRL-O I***
最適フィルモードのオン/オフ	***CTRL-O F***
タブモードのオン/オフ	***CTRL-O T***
逆インデントモードのオン/オフ	***CTRL-O U***
探索	***CTRL-Q F***
探索置換	***CTRL-Q A***
プレースマーカを探索	***CTRL-Q N***
前回の探索（置換）を繰り返す	***CTRL-L***
メニュー/エディタの切り換え	***F10***
ファイルをロードする	***F3***
セーブして編集を継続	***CTRL-K S*** または ***F2***
セーブせずにエディタから出る	***CTRL-K D*** または ***CTRL-K Q***
プレースマーカをセット	***CTRL-K 0*** ~ ***CTRL-K 3***
行の回復	***CTRL-Q L***
ペアマッチ（前方向）	***CTRL-Q]***
ペアマッチ（逆方向）	***CTRL-Q]***

基本的なカーソル移動コマンド

Turbo C エディタは，画面上でカーソルを上下左右に動きまわらせるには ***CTRL-文字***コマンドを使います。画面に表示されているファイルの一部分の中でカーソルを動かすには，つぎに示すような ***CTRL-文字***コマンドを使います。

キー	**動作**
CTRL-A	カーソルの左の単語の先頭文字へ移動
CTRL-S	カーソルを1文字左へ移動
CTRL-D	カーソルを1文字右へ移動
CTRL-F	カーソルの右の単語の先頭文字へ移動
CTRL-E	カーソルを1行上へ移動
CTRL-R	カーソルを1画面上へ移動
CTRL-X	カーソルを1行下へ移動
CTRL-C	カーソルを1画面下へ移動
CTRL-W	画面を1行下にスクロール（カーソルは同じ行にそのままある）

CTRL-Z 画面を1行上にスクロール(カーソルは同じ行にそのままある)
ROLLDOWN 1画面上にスクロール(カーソルも1画面上がる)
(IBM PC では ***PgUp***)
ROLLUP 1画面下にスクロール(カーソルも1画面下がる)
(IBM PC では ***PgDn***)

クイックカーソル移動コマンド

カーソルを素早く動かすコマンドも6個あります。これを使うと，カーソルを行の初めや終わり，ファイルの最初や最後，画面の最初や最後に即座に移動することができます。

CTRL-Q S または ***HOME***	カーソルを現在の行の先頭に移動
CTRL-Q D または ***SHIFT-HOME***(***End***)	カーソルを現在の行の最後に移動
CTRL-Q E	カーソルを画面の最上行に移動
CTRL-Q X	カーソルを画面の最下行に移動
CTRL-Q R	カーソルをファイルの先頭に移動
CTRL-Q C	カーソルをファイルの最後に移動

CTRL-Q の後に ***B***,***K***,***P*** などを続けると，ファイルの特定の場所にジャンプすることができます。

CTRL-Q B ***CTRL-K B*** でセットしたブロックの始点マーカにカーソルを移動します。このコマンドは，ブロックが表示されていなくても（ブロックコマンドの「**ブロックの表示/非表示**」の項を参照），あるいはブロックの終点マーカが設定されていなくても機能します。

CTRL-Q K ***CTRL-K K*** でセットしたブロックの終点マーカにカーソルを移動します。このコマンドも，ブロックが表示されていなくても，あるいはブロックの始点マーカが設定されていなくても機能します。

CTRL-Q P 直前に実行したコマンドの前のカーソル位置に移動します。このコマンドは，探索や探索/置換のコマンドを実行した後で，その実行の前の位置に戻りたい場合などに便利です。

挿入および削除コマンド

カーソル移動のコマンドしか知らないのでは，プログラムを書くには不充分です。テキストを挿入したり，削除できなくてはなりません。次に示すコマンドを使えば，文字，単語，行を挿入したり，削除することができます。

挿入モードのオン/オフ　　*CTRL-V* または *INS*

テキストを入力する場合には，**挿入モード**(Insert)あるいは**上書きモード**(Overwrite)のどちらかを選ばなくてはなりません。***CTRL-V*** あるいは ***INS*** キーを打つたびにモードが入れ替わります。現在のモードは，画面の一番上のステータス行に表示されています。

Turbo C エディタでは，挿入モードがデフォルトです。挿入モードでは，現在のテキストの中に新しい文字を挿入することができます。文字を入力していくにつれて，カーソルの右側にある文字列は右へ移動していきます。上書きモードでは，現在あるテキストは入力する新しい文字に置き換わっていきます。

カーソルの左の文字を削除/逆インデント　　*CTRL-H* または *BS*

カーソルを1文字左へ移動し，そこにある文字を削除します。カーソルの右側にある文字は1桁ずつ左へ移動します。このコマンドを使うと，2つの行の区切りを取り除くことができます。たとえば，行の先頭でこのコマンドを使うと，その行を1つ上の行の最後につなげることになります。

逆インデントモードがオンの場合，行の最初の非空白文字あるいは空白行にカーソルがあるときにこのコマンドを使うと，そこでのインデントが1レベル戻され，その間の文字が削除されます。

カーソル位置の文字の削除　　*CTRL-G* または *DEL*

カーソルの位置にある文字を削除し，カーソルの右側にある文字は1桁ずつ左に移動します。このコマンドで行の区切りを削除することができます。

カーソルの右の単語の削除　　*CTRL-T*

カーソルの右にある単語を削除します。単語というのは，次に示す文字のいずれかで区切られている文字の連なりと定義されます。

スペース　<　>　,　;　.　(　)　[　]　^　'　*　+　-　/　$

このコマンドで行の区切りを削除することができます。

行の挿入　　CTRL-N

カーソルの位置に行の区切りを挿入します。

行の削除　　CTRL-Y

カーソルのある行を削除し，それ以降の行を1行ずつ上に持ちあげます。一回消した行は回復できないので，このコマンドは注意して使ってください。

行の終わりまで削除　　CTRL-Q Y

現在のカーソルの位置から行の終わりまでのテキストを削除します。

ブロックコマンド

ブロックコマンドの場合もCTRL-文字を使います。ブロックは，テキストのひとかたまりであれば何でもよく，文字1個だけでも，数百行にわたるテキストであってもかまいません。ブロックを指定するにはブロックマーカを使います。ブロックは，ある時点では1個しか存在できません。

ブロックをマークするには，その先頭文字の前にブロック始点マーカ，最後の文字の後ろにブロック終点マーカをセットします。マーカをセットすると，そのブロックをコピーしたり，移動したり，ファイルに書き出したりすることができます。

ブロックの始点のマーク　　CTRL-K B

ブロックの始点をマークします。マーカそのものは見えません。始点と終点の両方のマーカをセットすると，ブロックが見えるようになります。マークされたブロックは，他のテキスト部分と区別できるようにハイライト表示されます。

ブロックの終点のマーク　　CTRL-K K

ブロックの終点をマークします。マーカそのものは見えません。始点と終点の両方のマーカをセットすると，ブロックが見えるようになります。

単語のマーク　　CTRL-K T

1個の単語をブロックとしてマークします。ブロック始点/終点マーカを使う必要はありません。カーソルが単語の中に位置していれば，その単語がマークされ，単語の中に位置していなければ，カーソルの左にある単語がマークされます。

ブロックのコピー　　CTRL-K C

マークされているブロックを，現在カーソルがある場所にコピーします。元のブロック内のテキストはそのままで，マーカは新しくコピーされた場所に移動します。マークされているブロックがなかったり，マークされたブロックの中にカーソルがある場合は，何も行なわれません。

ブロックの削除　　CTRL-K Y

マークされているブロックを削除します。一度消してしまうと回復する手立てはないので，このコマンドは注意して使用してください。

ブロックの表示/非表示　　CTRL-K H

マークされているブロックを表示するか，非表示にするかを決めます。このコマンドを入力するたびに，表示/非表示が切り換わります。ブロック処理コマンド（コピー，移動，削除，ファイルへの書き出し）は，ブロックが表示されている場合しか働きません。ブロックに関連するカーソル移動（ブロックの先頭/最後へのジャンプ）は，ブロックが非表示の場合にも機能します。

ブロックの移動　　CTRL-K V

マークされたブロックをカーソルの位置に移動します。元の場所のブロックは無くなります。マーカも新しい位置に移動します。マークされたブロックが存在しない場合は何も行なわれません。

ディスクからのブロックの読み込み　　CTRL-K R

ディスク上にあるファイルをカーソルの位置に読み込みます。読み込まれたテキストはブロックとなり，ハイライト表示されます。

このコマンドを入力すると，読み込むファイルの名前を入力するためのボックスが表示されます。ここでワイルドカードを使うと，小さなウィンドウが画面に現われ，ディレクトリが表示されます。指定するファイル名は，正しいものであればどんなものでもかまいません。拡張子（.C，.TXT，.BAK など）を指定しないと，.C とみなされます。拡張子のないファイルを読み込むには，ファイル名の後ろにピリオドをつけな

くてはなりません。

ディスクへのブロックの書き出し　　***CTRL-K W***

マークしたブロックをディスクに書き出します。ブロックはそのままで，マーカも元のままです。マークされたブロックがない場合は何も行なわれません。

このコマンドを入力すると，書き出すファイルの名前を入力するためのボックスが表示されます。ワイルドカードを指定すると，画面に小さなウィンドウが現われ，ディレクトリが表示されます。指定したファイルが既に存在する場合は，エディタは警告メッセージを出力し，実際に上書きを行なう前に確認を求めてきます。ファイル名は，正しいものであればどんなものでもかまいません(拡張子のデフォルトは.C です)。拡張子を持たないファイルを指定するには，ファイル名の後ろにピリオドをつけなくてなりません。

プリンタへのブロックの印字　　***CTRL-K P***

マークしたブロックをプリンタに印字します。***CTRL-K W*** で，ファイル名に PRN を指定してもプリンタに印字することができますが，その場合タブ文字 (0x09) はそのままプリンタに送られます。***CTRL-K P*** を使うと，タブ文字は画面に表示されている分の空白に置き換えてからプリンタに送られます。ブロックが定義されていない場合には，ファイル全体がプリントされます。

その他の編集コマンド

この節では，これまで述べてきたカテゴリに入らないコマンドについて説明します。

操作の中止　　***CTRL-U***

進行中のコマンドが入力を求めてきたとき，たとえば探索/置換コマンドで Replace (Y/N)? と聞いてきたり，探索文字列を聞いてきた段階で，そのコマンドの実行を中止します。

自動インデントのオン/オフ　　***CTRL-O I***

引き続く行の自動インデントを行なうかどうかを決めます。このコマンドを入力するごとにオン/オフが切り替わります。自動インデントがオンのときには，**リターン**キーを押すと，カーソルは次の行の先頭ではなく，1つ上の行で空白またはタブ以外の最初の文字と同じ桁に位置します。

インデントの桁位置を変えたい場合は，スペースあるいは ***TAB***，および左矢印キーあるいは ***BS*** キーを使って希望する位置にカーソルを動かせばよいわけです。自動インデントがオンのときは，ステータス行に Indent という表示が出ています。オフにすると表示は消えます。自動インデントのデフォルトはオンです。

制御文字の入力　　***CTRL-P***

ファイルの中に制御文字を入力するには，まず ***CTRL-P*** を入力してから制御文字を入力します。制御文字は低輝度で（あるいは反転文字で）大文字で表示されます。

探索　　***CTRL-Q F***

30文字までの文字列を探索します。このコマンドを入力すると，ステータス行が消えて Find：というプロンプトが表示され，探索する文字列を聞いてきます。探索したい文字列を入力して**リターン**を押します。

探索文字列には，制御文字も含めてどんな文字でも入れることができます。探索文字列に制御文字を入力する場合は，前に ***CTRL-P*** をつけて入力します。たとえば，***CTRL-T*** を入力するには，***CTRL*** キーを押しながら，***P T*** の順でキーを押せばよいことになります。探索文字列の中に行区切りを含めるには，***CTRL-M CTRL-J***（復帰/改行）を入力します。また，***CTRL-A*** は特別な意味を持っており，これはどんな文字ともマッチします。たとえば，X^AZ は XYZ や X1Z にマッチします。

1度探索が行なわれている場合，探索コマンドを呼び出すと，FIND：の入力エリアに前回の探索文字列が表示されます。探索文字列の入力エリアの中では，カーソル移動キーは次のような動作をします。

CTRL-S，***CTRL-H***，***BS***
　　カーソルを1字左に動かして，カーソルの左の文字を消す。

CTRL-D　カーソル位置に，前回の文字列のその位置の文字を表示し，カーソルを右に1文字動かす。

CTRL-A　入力エリア全体をクリアし，カーソルを入力エリアの最初に動かす。

CTRL-F　カーソル位置に，前回の文字列のその位置以降の文字を表示し，カーソルを文字列の終わりに動かす。

探索文字列を入力すると，Turbo C エディタは Options：を表示して探索オプションを聞いてきます。オプションには次のようなものがあります。

B　　現在のカーソルの位置からファイルの先頭に向かって，逆向きに探索を

行ないます。

N　現在のカーソルの位置からスタートして，次に現われる探索文字列にカーソルを位置づけます。

n　*n*は整数で，現在のカーソルの位置からスタートして，*n*番目に見つかった探索文字列にカーソルを位置づけます。

U　大文字小文字の区別をせずに探索します。

W　1つの単語となっているものだけを探索します。つまり，他の単語の一部分の場合にはスキップします。

探索コマンドオプションの例：

W　完全な単語のみを探索します。探索文字列termは，terminalとは一致しないことになります。

BU　逆向きに探索し，大文字小文字の区別はしません。Blockは，BLOCKADEともblockheadとも一致します。

125　探索文字列で，125番目に現われるものを探索します。

オプションを入力して（何も指定しない場合もありますが）**リターン**を押すと，探索が開始されます。探索文字列と一致するものがテキストの中に見つかると，エディタはそこにカーソルを位置づけます。この探索操作は，探索繰り返しコマンド（***CTRL-L***）で繰り返すことができます。

探索置換　***CTRL-Q A***

このコマンドは，探索コマンドとほぼ同じですが，見つかった文字列を30文字以内の文字列で置き換えることができます。***CTRL-A***は，探索文字列の中ではワイルドカードとして使えますが，置換文字列の中では特殊な意味は持ちません。

探索文字列を指定すると，エディタは置換文字列を聞いてきます。30文字以内の文字列を入力します。制御文字の入力や編集の手順は，探索コマンドの場合と同じです。

置換文字列を入力せずに**リターン**キーを押すと，見つけた文字列を削除することになります。

オプションは探索コマンドの場合とほぼ同じですが，次の4つが追加されています。

G　現在のカーソルの位置に関係なく，ファイル全体に対して探索/置換します。カーソルは最後に見つかった探索文字列に位置づけられます。

L　マークされたブロックの中でのみローカルに探索/置換を行ないます。

N　問い合わせをせずに置き換えます。探索文字列を見つけても，いちいち置

き換えるかどうかの確認をしないのです。NとG両方を指定した場合は，Gが有効になります。

n 次の *n* 個の探索文字列を置き換えます。Gオプションが指定されている場合は探索はファイルの先頭からスタートし，そうでない場合は現在のカーソルの位置からスタートします。

探索/置換コマンドオプションの例：

N10 カーソル以降で探索文字列に一致するものを10回，問い合わせをせずに置き換えます。

GW ファイル全体に対して，完全な単語で探索文字列に一致するものを探します。大文字小文字の区別はしません。置き換えるかどうかの問い合わせは行ないます。

GNU ファイル全体に対して，大文字小文字を区別せずに探索を行ない，問い合わせをせずに置き換えます。

オプションを入力して（何も入力しないときもありますが）**リターン**キーを押すと，探索/置換操作がスタートします。探索文字列に一致するものが見つかると，(Nオプションを指定していなければ)その最後にカーソルを位置づけ，Replace (Y/N)?というプロンプトを画面一番上の行に表示するので，これに答えます。ここで ***CTRL-U*** を入力すれば，このコマンドの実行を中止することもできます。探索/置換の操作は，***CTRL-L*** で繰り返すことができます。

プレースマーカの探索　***CTRL-Q 0, CTRL-Q 1, CTRL-Q 2, CTRL-Q 3***

4個までのプレースマーカ（0〜3）を探索します。***CTRL-Q*** の入力の後，マーカ番号 *n* を入力すると，前にセットされたマーカにカーソルを移動することができます。

ファイルのロード　***F3***

既存のファイルを編集するか，新規ファイルを作成することになります。

最適フィルモードのオン/オフ　***CTRL-O F***

タブモードがオンのときに最適フィルモードがオンになっていると，エディタは各行の先頭の空白部分を，タブ文字（0x09）と空白文字（0x20）によって最適に（バイト数が最も少なくなるように）埋めます。

エディタを出る（セーブはしない）　***CTRL-K D*** または ***CTRL-K Q***

エディタを出てメインメニューに移ります。編集の済んだファイルをディスクにセーブするには，メインメニューから File コマンドの Save オプションを選択するか，エディタの中で ***CTRL-K S*** または ***F2*** を押します。

直前の探索の繰り返し　　*CTRL-L*

直前に行なわれた探索または探索/置換コマンドを繰り返します。

行の回復　　*CTRL-Q L*

次の行に移ってしまわなければ，その行に対する変更を取り消すことができます。行なった変更がどんなものであっても，行は元の状態に戻ります。ただし，行を削除してしまったり，***DEL*** や ***BS*** キーで行の区切りを削除してしまった場合には，これを元に戻すことはできません。

ファイルのセーブ　　*CTRL-K S* または *F2*

編集中のファイルをディスクにセーブします。エディタからは出ないので編集は継続できます。

プレースマーカの設定　　*CTRL-K 0, CTRL-K 1, CTRL-K 2, CTRL-K 3*

テキスト内で，4個までの位置をマークすることができます。***CTRL-K*** に続いて0～3の数字を入力します。マークを行なった後は，ファイルの中のどこにいても，***CTRL-Q*** *n* を入力すれば（*n* は0～3の数字），マークしてある位置に容易に戻ることができます。

タブ　　*CTRL-I* または *TAB*

タブモードがオンの場合には，タブ文字（0x09）を挿入し，次のタブ位置にカーソルを移動します。タブ位置は，デフォルトでは8桁おきにセットされています。タブの桁数は，Options/Environment メニューの中の Tab size オプションで，2～16の範囲で設定することができます。

タブモードがオフの場合には，1つ上の行の単語の頭文字と同じ桁にカーソルが移動します。この場合にはタブ文字は書き込まれず，必要な数だけ空白（0x20）が挿入されます。

タブモード　　*CTRL-O T*

ステータス行に Tab と表示されているときには，タブモードがオンになっていることを意味します。このコマンドで，タブモードのオン/オフが切り換わります。

逆インデントのオン/オフ　　*CTRL-O U*

逆インデントモードのオン/オフを切り換えます。逆インデントモードがオンの場合，カーソルを行の最初の非空白文字あるいは空白行に置いて ***BS*** キー（または ***CTRL-H***）を押すと，そこでのインデントが1レベル戻されます。

ペアマッチ　　*CTRL-Q* [（前向き）　CTRL-Q]（逆向き）

カッコやコメントの始まりと終わりの対応を調べます。***CTRL-Q*** [は前向き（ファイルの最後に向かって）探索します。***CTRL-Q***]は逆向き（ファイルの先頭に向かって）探索します。

対応を調べることができるのは，次の7種類のペアです。

{ }　< >　( )　[ ]　/* */　" "　' '

対応を調べたい区切り文字にカーソルを置いてコマンドを実行します。4種類のカッコとコメントの区切り（/＊　＊/）については，対応するもう一方がどちらの方向にあるかは明らかであり，どちらのコマンドを使っても正しく対応が調べられます。2種類の引用符については，エディタには方向がわからないので，どちら向きに対応する引用符を探すのかを，コマンドによって区別しなければなりません。なお，コメントの区切りについては，2文字のうちの1文字目（'/＊'の'/'，'＊/'の'＊'）にカーソルを置いてコマンドを実行してください。

ペアマッチについては，第5章の第 IV 部で使い方を詳しく説明しています。

Turbo C エディタ vs. WordStar

Turbo C エディタのコマンドのいくつかは，WordStar とは少し違っています。Turbo C エディタのコマンドは WordStar のサブセットですが，ソースコードの編集を効率よく行なえるようにするために，WordStar にはない機能をいくつか持っています。こうした相違点についてここで説明しておきます。

自動インデント：

Turbo C エディタの ***CTRL-O I*** コマンドは，自動インデントのオン/オフを行ないます。

カーソルの移動：

Turbo C エディタのカーソル移動コマンド（***CTRL-S***，***CTRL-D***，***CTRL-E***，***CTRL-X***）

を使うと，画面上で自由にカーソルを移動させることができます。空白行でも，カーソルが1桁目にきてしまうことはありません。だからといって，ファイルが空白文字で埋まっているわけではなく，行の最後の文字より後ろの空白は自動的に取り除かれます。カーソルをこのように自由に動かせることは，プログラムを編集する際（たとえば，文のインデントを揃えるときなど）にはとても便利です。

カーソル位置から左の削除：

WordStar では，***CTRL-Q DEL*** と押すとカーソルの位置から行の初めまでを消すことができますが，この機能はサポートされていません。

単語をブロックとしてマーク：

Turbo C では ***CTRL-K T*** コマンドを使えば，1個の単語をブロックとしてマークすることができます。WordStar では，単語をマークする場合にも，単語の先頭と最後をマークする2ステップが必要です。

行の区切りを越えての動き：

CTRL-S や ***CTRL-D*** では，行の区切りを越えてカーソルを動かすことはできません。別の行に移るには，***CTRL-E***，***CTRL-X***，***CTRL-A***，***CTRL-F*** を使うことになります。

エディットの終了：

Turbo C の ***CTRL-K Q*** は，WordStar の ***CTRL-K Q***（編集終了）とは異なります。Turbo C では，変更したテキストは保存されており，引き続いてコンパイルすることもセーブすることもできます。

アンドゥ：

Turbo C の ***CTRL-Q L*** コマンドを使うと，他の行に移る前であれば，行に対して行なった変更を取り消すことができます。

ディスクファイルの更新：

Turbo C での編集はほとんどメモリ上で行なっているので，WordStar のように ***CTRL-K D*** ではディスクには書き込まれません。したがって，File メニューの Save オプションを使うか，エディタ内で ***CTRL-K S*** または ***F2*** を押してディスクに書き込まなければなりません。

付録B

TCC コマンドラインオプション

この付録では，Turbo C のコンパイル時のコマンドラインオプションを，オプションのタイプでわけ，それぞれの中ではABC順に並べて示し，各オプションがどのようなことをするのかについて述べています。オプションは次の3種類に分類することができます。

■コンパイラオプション
■リンカオプション
■環境オプション

コンパイラオプションは，さらに次のようなカテゴリにわけることができます。

■メモリモデルオプション
■# defines（マクロ定義オプション）
■コード生成オプション
■最適化オプション
■ソースコードオプション
■エラーオプション
■セグメント名の制御

DOS のプロンプトから tcc だけをタイプしてリターンを押すと，TCC(コマンドライン版の Turbo C）のすべてのオプションの要約が表示されます。

コマンドラインオプションの多くのものは，Turbo C 統合開発環境の Options メニューに対応するものを持っています（他のメニューにも対応するものが少し含まれています)。統合環境のメニュー項目とコマンドラインオプションの対応表を，**表 2.2** に示してあります。

表 2.2　コマンドラインオプションとメニュー項目の対応関係

コマンドラインオプション	TC でのメニュー項目
-A	**O**/**C**/**S**ource/**A**NSI keywords only...On
-a	**O**/**C**/**C**ode generation/**A**lignment...Word
-a-　＊＊	**O**/**C**/**C**ode generation/**A**lignment...Byte
-B	(対応するメニュー項目はありません)
-C	**O**/**C**/**S**ource/**N**ested comments...On
-c	**C**ompile/**C**ompile to OBJ
-D*name*[=*string*]	**O**/**C**/**D**efines
-d	**O**/**C**/**C**ode generation/**M**erge duplicate strings...On
-d-　＊＊	**O**/**C**/**C**ode generation/**M**erge duplicate strings...Off
-E*filename*	(対応するメニュー項目はありません)
-e*filename*	**P**roject/**P**roject Name
-f　＊＊	**O**/**C**/**C**ode generation/**F**loating point...Emulation
-f-	**O**/**C**/**C**ode generation/**F**loating point...None
-f87	**O**/**C**/**C**ode generation/**F**loating point...8087
-G	**O**/**C**/**O**ptimization/**O**ptimize for ...Speed
-g #	**O**/**C**/**E**rrors/**W**arning：stop after...#
-I*pathname*	**O**/**D**/**I**nclude directories
-i #	**O**/**C**/**S**/**I**dentifier length...#
-J　＊＊	**O**/**C**/**C**ode generation/**K**anji strings...On（PC-9801のみ）
-J-	**O**/**C**/**C**ode generation/**K**anji strings...Off（PC-9801のみ）
-j #	**O**/**C**/**E**rrors/**E**rrors：stop after...#
-K	**O**/**C**/**C**ode generation/**D**efault char type...Unsigned
-K-　＊＊	**O**/**C**/**C**ode generation/**D**efault char type...Signed
-k	**O**/**C**/**C**ode generation/**S**tandard stack frame...On
-L*pathname*	**O**/**D**/**L**ibrary directories
-lx	(対応するメニュー項目はありません)
-M	**O**/**L**/**M**ap file
-mc	**O**/**C**/**M**odel...Compact
-mh	**O**/**C**/**M**odel...**H**uge
-ml	**O**/**C**/**M**odel...**L**arge
-mm	**O**/**C**/**M**odel...**M**edium

-ms **	**O**/**C**/**M**odel...Small
-mt	**O**/**C**/**M**odel...**T**iny
-N	**O**/**C**/**C**ode generation/**T**est stack overflow...On
-n*pathname*	**O**/**D**/**O**utput directory
-O	**O**/**C**/**O**ptimization/**J**ump optimization...On
-o*filename*	(対応するメニュー項目はありません)
-P	**O**/**C**/**C**ode generation/**C**alling convention...Pascal
-P- **	**C**/**C**/**C**ode generation/**C**alling convention...C
-r **	**O**/**C**/**O**ptimization/**U**se register variables...On
-S	(対応するメニュー項目はありません)
-U*name*	(対応するメニュー項目はありません)
-u **	**O**/**C**/**C**ode generation/**G**enerate underbars...On
-v	**D**ebug/**S**ource debugging...On
-w	**O**/**C**/**E**rrors/**D**isplay warnings...On
-w-	**O**/**C**/**E**rrors/**D**isplay warnings...Off
-w*xxx*	**O**/**C**/**E**rrors/**P**ortability warnings, **A**NSI violations, **C**ommon errors, **L**ess common errors...On
-w-*xxx*	**O**/**C**/**E**rrors/**P**ortability warnings, **A**NSI violations, **C**ommon errors, **L**ess common errors...Off
-y	**O**/**C**/**C**ode generation/**L**ine numbers...On
-Z	**O**/**C**/**O**ptimization/**R**egister optimization...On
-zA*name*	**O**/**C**/**N**ames/**C**ode/**C**lass
-zB*name*	**O**/**C**/**N**ames/**B**SS/**C**lass
-zC*name*	**O**/**C**/**N**ames/**C**ode/**S**egment
-zD*name*	**O**/**C**/**N**ames/**B**SS/**S**egment
-zG*name*	**O**/**C**/**N**ames/**B**SS/**G**roup
-zP*name*	**O**/**C**/**N**ames/**C**ode/**G**roup
-zR*name*	**O**/**C**/**N**ames/**D**ata/**S**egment
-zS*name*	**O**/**C**/**N**ames/**D**ata/**G**roup
-zT*name*	**O**/**C**/**N**ames/**D**ata/**C**lass
-1	**O**/**C**/**C**ode generation/**I**nstruction set...80186/80286
-1- **	**O**/**C**/**C**ode generation/**I**nstruction set...8088/8086

O/=**O**ptions **C**/=**C**ompiler **E**/=**E**nvironment **D**/=**D**irectories **=デフォルト

オプションのオン/オフの指定

コマンドラインオプションの選択は，まずマイナス記号を入力し，すぐ続いてオプション文字を入力して行ないます(-I など)。オプションをオフにするには，オプション文字の後にもうひとつマイナス記号を入力します。たとえば，-A は ANSI キーワードオプションをオンにしますが，-A- とするとこのオプションはオフとなるわけです。

こうして簡単にコマンドラインで各スイッチのオン/オフが行なえます。こうして指定したものは，コンフィギュレーションファイル中での設定に優先します。

構文

MS-DOSのコマンドライン上でTurbo Cのコンパイルオプションを選択するには，次のような構文で行ないます。

```
tcc [option option ...] filename filename ...
```

Turbo Cは次に示す規則に従って，ファイルをコンパイルしていきます。

filename.**asm**	TASMを呼び出してアセンブルし，.OBJファイルを作る
filename.**obj**	リンク時にオブジェクトとして取り込む
filename.**lib**	リンク時にライブラリとして取り込む
filename	*filename*.**c**をコンパイルする
filename.**c**	*filename*.**c**をコンパイルする
filename.**xyz**	*filename*.**xyz**をコンパイルする

たとえば，次のようなコマンドラインの場合

```
tcc -a -f -c -o -Z -emyexe oldfile1.c oldfile2 nextfile.c
```

TCCはOLDFILE1.C, OLDFILE2.C, NEXTFILE.Cをコンパイルして，.OBJファイルを作り出し，その後MYEXE.EXEという名前の実行可能なプログラムファイルを作ります。その際，ワード境界割り付け(-a)，浮動小数点演算エミュレーション(-f)，コメントのネスト（-c)，ジャンプの最適化（-o)，レジスタ最適化（-Z）のオプションを選択しています。

コマンドラインで.ASMファイルを与えているか，.Cファイルの中にインラインアセンブリが含まれている場合，TCCはTASMを呼び出します。TCCはTASMに次のスイッチを渡します。

```
/mx /D__mdl__
```

この中の*mdl*は，TINY，SMALL，MEDIUM，COMPACT，LARGE，HUGEのうちのいずれかです。/mxスイッチは，MASMに大文字小文字を区別してアセンブルを行なうように指定するためのものです。

コンパイラオプション

Turbo Cのコンパイラオプションは8つのグループにわけられます。これらの概要を説明します。

- **メモリモデルオプション**では，どのメモリモデルでコンパイルするかを指定します(メモリモデルは，タイニィ，スモール，ミディアム，コンパクト，ラージ，ヒュージです)。
- **# defines（マクロ定義）**は，シンボリック定数などのマクロをコマンドライン上で定義するためのものです。デフォルトの定義は空白文字1個ですが，数値や文字列を指定することもできます。また，定義済みのマクロを未定義にすることもできます。
- **コード生成オプション**は，実行時に使用されるコードの生成に関して指示を与えるものです。これには，浮動小数点演算のモード，文字型，CPUの命令セットなどがあります。
- **最適化オプション**では，オブジェクトコードをどのように最適化するかを指示します。速度とサイズのどちらを優先して最適化するか，レジスタ変数の使用の有無，無駄なロードの削除をするかどうかなどを決めることができます。
- **ソースコードオプション**は，ソースコードに関する特定の機能を有効（または無効）にします。ここで指示するものには，処理系固有のキーワード（ANSI規格案でない），コメントのネスト，識別名の長さなどがあります。
- **エラーオプション**では，どのような警告メッセージをコンパイラが出すか，最大いくつの警告やエラーが起きたらコンパイルをストップするかを指示します。
- **セグメント名オプション**では，セグメント名を変えたり，そのグループやクラスを変えることができます。
- **コンパイル制御オプション**では，コンパイラに以下の指示を出します。
  - ・アセンブラコードを出力する（オブジェクトコードは生成しない)。
  - ・インラインアセンブルを行なう。
  - ・リンクせずにコンパイルだけ行なう。

メモリモデルオプション

-mc　コンパクトメモリモデルを使ってコンパイルします。

-mh　ヒュージメモリモデルを使ってコンパイルします。

-ml　ラージメモリモデルを使ってコンパイルします。

-mm　ミディアムメモリモデルを使ってコンパイルします。

-ms　スモールメモリモデルを使ってコンパイルします。

-mt　タイニィメモリモデルを使ってコンパイルします。スモールメモリモデルとほぼ同じコードを生成しますが，タイニィモデルのプログラムをリンクするときにはC0T.OBJ を使用します。

Turbo C のメモリモデルの詳細については第12章を参照してください。

defines（マクロ定義）

-D*xxx*　識別名 *xxx* を1個または複数の空白文字からなる文字列として定義します。

-D*xxx*[＝***string***]
指定した識別名 *xxx* に，等号の後ろの文字列 *string* を定義します。*string* の中にはスペースやタブを入れることはできません。

-U*xxx*　識別名 *xxx* に関する，前に行なわれた定義を未定義にします。

Turbo C では，次の3つの方法で複数の# define マクロをコマンドライン上で定義することができます。

■1つの-D オプションの後ろに，セミコロンで区切って複数の定義を置く。

```
tcc -Dxxx;yyy=1;zzz=NO myfile.c
```

■コマンドラインに複数の-Dオプションを置く。

```
tcc -Dxxx -Dyyy=1 -Dzzz=NO myfile.c
```

■上の2つを組み合わせる。

```
tcc -Dxxx -Dyyy=1;zzz=NO myfile.c
```

コード生成オプション

-1 Turbo Cに，80186の拡張命令を生成するように指示します。このオプションは，MS-DOSのもとでのIBM PC/ATなどで，リアルモードで走る80286プログラムを生成する場合にも使用されます。

-a 整数の大きさの領域を占める項目をワード境界で始まるようにします。構造体の場合には，必要ならば余分なバイトを埋め込んで，メンバがワード境界で始まるように割り当てられます。自動変数とグローバル変数は正しく割り当てられます。charやunsigned char型の変数およびメンバはどんなアドレスにも置くことができます。これ以外のものは，すべて偶数アドレスに置かれなければなりません。デフォルトはオフで，バイトアラインメントになります。

-d ある文字列が他の文字列と一致する場合に，それらをマージします(1つの文字列にする)。こうすると，プログラムサイズを小さくすることができます(オフがデフォルトです)。

-f87 浮動小数点演算を，8087エミュレーションライブラリルーチンを使うのではなく，インラインの8087命令を使うようにコードを生成します。浮動小数点コプロセッサが実行時には利用できることを意味しているので，このオプションでコンパイルしたプログラムは，浮動小数点コプロセッサが載っていないマシンでは走りません（現行バージョンでは，このオプションはどのライブラリがリンクされるかについてのみ影響します)。

-f 実行時のシステムが8087を搭載していない場合は8087呼び出しをエミュレートし，8087が搭載されてる場合は，浮動小数点演算は8087を呼び出して行ないます（デ

フォルトはオンです)。

-f- プログラムが浮動小数点演算を含んでいないことを指定します。したがって，リンク時に浮動小数点ライブラリはリンクされません。

-J コンパイラに漢字文字列を正しく認識するよう指示します。これによって，ソーステキスト中の漢字コードの第2バイトが，エスケープ文字（¥），ダブルクォート（"），シングルクォート（'）と同じコードを持つ漢字は，文字列の一部として正しく扱われます（デフォルトはオンです)。

-J- プログラムが漢字を含んでいないことを指定します。もしこのオプションを指定したコンパイルしたプログラムに含まれている漢字コードは，グラフィック文字として認識されます。

-K コンパイラに，char 型の宣言すべてを unsigned char 型として扱うように指示します。これで，char 型の宣言を unsigned として扱う他のシステムとの互換性を取ることが可能となります。デフォルトでは，char 型の宣言は signed として扱われます。

-k デバッガがスタックを通してサブルーチンコールのトレースバックをしやすくするために，標準のスタックフレームを生成します。デフォルトはオンです。

-N 各関数のエントリ（入口）に，スタックオーバーフローのチェック機構を埋め込みます。このチェック機構がスタックオーバーフローを検出すると，それを示すメッセージが表示されます。チェック機構を組み込むと，プログラムサイズと実行速度は犠牲になりますが，スタックオーバーフローは非常に見つけにくいので，このオプションは開発段階ではたいへん有用です。オーバーフローが検出されると，Stack overflow というメッセージを表示して，プログラムは終了コード1で異常終了します。

-p コンパイラに，すべてのサブルーチン呼び出しおよびすべての関数を，Pascal の引数受け渡し手順を使って生成することを指示します。こうすると，関数呼び出しのコードは小さくかつ速くなります。ただしこの場合，C の手順とは違って，引数の型と数は正しく対応させなければならず，可変個の引数も使用できません。このオプションを指定していても，cdecl 文を使えば，特定の関数についてはオプ

ションを無視してCの手順を適用することができます。

-u 宣言された識別名をオブジェクトモジュールに格納する際に，識別名の前に自動的に下線（_）を付加します。

Turbo Cは，pascal 型の識別名（予約語 pascal によって修飾される）に対しては別の扱いをします。これらは大文字化され下線は付加されません。

Cの識別名に対する下線の付加はオプションですが，デフォルトではオンです。-u-とすればこれをオフにできます。しかし Turbo Cの標準ライブラリを使うのであれば，ライブラリの再構築をしない限り問題を生じます（ライブラリの再構築には実行時ライブラリのソースコードが必要です。詳しくは MSA までお問い合わせください)。

下線についての詳細は第12章を参照してください。

注意：エキスパート以外は-u-オプションは使用しないでください。

-y シンボリックデバッガで使用する行番号を，オブジェクトファイルの中に埋め込みます。こうすると，オブジェクトファイルは大きくなりますが，実行可能プログラムの大きさや実行スピードには影響を与えません。このオプションは，シンボリックデバッガと併用する場合にのみ意味があります。

-v コンパイラに.OBJ ファイルの中にデバッグ情報を含めるように指示します。これによって TC統合デバッガあるいはスタンドアロンのデバッガによるデバッグが可能となります。コンパイラはこのスイッチをリンカにも渡すので，.EXE ファイルの中にもデバッグ情報が入ります。

最適化オプション

-G コンパイラに，サイズの小さいルーチンよりは，実行スピードの速いルーチンを作るように指示します。

-O 重複したジャンプを削除したり，ループや switch 文を再編成することによって，コードサイズを減らす最適化を行ないます。

-r- レジスタ変数の使用を抑えます。

-r-オプションを指定した場合，あるいは **O/C/Optimization/Use register vari-**

ablesをオンにした場合，コンパイラはレジスタ変数をまったく使用せず，またそのルーチンを呼び出す側のレジスタ変数（SI,DI）を保存も想定もしません。したがって，レジスタ変数を使用するコード内から，-r-オプションでコンパイルしたルーチンを呼び出してはなりません。

また，SIとDIを保存しない既存のアセンブリ言語コードとのインターフェースをとりたいときには，この-r-オプションによって，そのコードをTurbo Cから呼び出すことができます。

注意：エキスパート以外は-r-オプションは使用しないでください。

-r　レジスタ変数の使用を可能とします（デフォルトです）。

-Z　レジスタの内容を記憶しておくことにより，無駄なレジスタへのロード操作をやめ，可能な限りそれらの値を使用します。

注意：このオプションを使う場合には注意が必要です。レジスタがポインタによって間接的に変更を受けるかもしれず，コンパイラはそれを検出することができないからです。

たとえば，変数 *A* がレジスタDXにロードされた場合，その値は保持されます。その後で *A* に値が代入されると，DXの値はもはや使用すべきではないのでリセットされます。もし，*A* の値が *A* を指すポインタを使っての代入によって間接的に変更された場合は，Turbo Cはこれを検出できず，DXは *A* の（すでに古くなった）値を含んでいると思い続けるのです。

-Zによる最適化は，すでにレジスタに入っている値を，もう一度ロードするのを抑えるように設計されたものです。これにより，ロード命令自体を削除できることもあるし，そうでなくてもメモリを参照する命令をレジスタを参照する命令に変更することができるのです。

次に示した少し人工的なコードは，この最適化の利点と欠点の両方を明らかにしており，-Zを使う場合はくれぐれも注意が必要であることを示しています。

Cのコード	最適化アセンブラ

```
func()
{
    int   A, *P, B;

    A = 4;                          mov   A,4
    ...
    B = A;                          mov   ax,A
                                    mov   B,ax
    P = &A;                         lea   bx,A
                                    mov   P,bx
    *P = B + 5;                     mov   dx,ax
                                    add   dx,5
                                    add   [bx],dx
    printf("%d¥n", A);              push  ax
}
```

まず＊P＝ B+5;という文に注目すると，生成されたコードは，ax から dx への mov 命令で始まっています。もしこれが-Z オプションを指定せずにコンパイルされた場合は，*B* からの mov，つまり長くて遅い命令が使われることになります。

次に，＊*P* への代入は，*P* がすでに bx の中に入っているので，add 命令の後で本来行なうべき *P* から bx への mov は省略することができます。この最適化は正しいものであり，一般に役立つものです。

しかし，**printf** で出力される値は正しくないはずです。Turbo C は ax が *A* の値を保持していると思っているので，メモリの内容を push せずに，ax レジスタの内容を push してしまいます。**printf** は，正しい値9ではなく4を表示するはずです。*P* を使った間接的な代入が *A* の値の変化を隠してしまったのです。

＊P ＝ B+5;という文が A ＝ B+5;と書かれていれば，Turbo C は値の変化を認識することができます。

レジスタの内容は，関数呼び出しやジャンプ(ラベル，case 文，ループの先頭や最後など）が行なわれた段階で変化しています。こうした制限と，8086プロセッサファミリィではレジスタの数が少ないことのために，この最適化を使ったプログラムの多くは，変な動きをすることはほとんどありません。

ソースコードオプション

-A　ANSIコンパチブルなソースコードをコンパイルします。Turbo Cで拡張された予約語は無視され，（おそらくは）通常の識別名として扱われます。このような予約語には以下のものがあります。

```
near        far         huge        cdecl
asm         pascal      interrupt
_es         _ds         _cs         _ss
```

およびレジスタ疑似変数，_AX，_BX，_SIなどです。

-C　コメントのネストを許します。コメントは通常ネストしてはいけないことになっています。

-i#　コンパイラは，識別名の先頭の#個の文字を認識します。すべての変数は，それが変数であるか，プリプロセッサマクロであるか，構造体のメンバ名であるかに関係なく，先頭の#文字が違っているときに，別のものであると認識されます。Turbo Cでは，#のデフォルト値は32です。UNIXも含めて他の多くのシステムでは，先頭から8文字までしか認識しません。このようなシステムでも使えるプログラムを開発する場合は，#を小さな値にしてコンパイルすればよいでしょう。こうすると，長い識別名の後ろが削られたとき，識別名の重複が起こるかどうかをチェックできるからです。

エラーオプション

-g#　#個のメッセージ（警告とエラーメッセージあわせて）が出力された段階でコンパイルをやめます。

-j#　#個のエラーメッセージが出力された段階でコンパイルをやめます。

-w*xxx* *xxx* によって指定される警告メッセージの出力を可能にします。

オプション-w-*xxx* は，*xxx* によって指定される警告メッセージの出力を抑制します。警告に関する詳細についてはリファレンスガイドの付録 A を参照してください。-w*xxx* に対して可能な値は次のとおりです。

■ **ANSI violations**（ANSI 規格違反）

-wbig *	Hexadecimal or octal too large. （16進または8進定数が大きすぎる）
-wdup *	Redefinition of 'XXXXXXXX' is not identical. （XXXXXXXX の再定義が同一でない）
-wret *	Both return and return of a value used. （値のない return と，値を伴う return が使われている）
-wstr *	'XXXXXXXX' not part of structure. （XXXXXXXX は構造体の一部分ではない）
-wstu *	Undefined structure 'XXXXXXXX'. （構造体 XXXXXXXX は未定義である）
-wsus *	Suspicious pointer conversion.（あやしげなポインタの変換）
-wvoi *	Void functions may not return a value. （void 型の関数は値を返せない）
-wzst *	Zero length structure.（長さゼロの構造体）

■ **Common errors**（よく起こるエラー）

-waus *	'XXXXXXXX' is assigned a value that is never used. （XXXXXXXX には値が与えられているが，1回も使われていない）
-wdef *	Possible use of 'XXXXXXXX' before definition （XXXXXXXX は定義される前に使われている）
-weff *	Code has no effect.（コードは何の効果も持っていない）
-wpar *	Parameter 'XXXXXXXX' is never used. （引数 XXXXXXXX は1度も使われない）
-wpia *	Possibly incorrect assignment.（おそらく誤った代入である）
-wrch *	Unreachable code.（このコードには制御は到着しない）
-wrvl	Function should return a value.（関数は値を返すべきである）

■ Less common errors（あまり起こらないエラー）

-wamb　Ambiguous operators need parentheses.
（任意性のある演算子であり，カッコが必要）

-wamp　Superfluous & with function or array.
（関数や配列に余分な&がついている）

-wnod　No declaration for function 'XXXXXXXX'.
（関数 XXXXXXXX の宣言がない）

-wpro　Call to function with no prototype.
（プロトタイプがない関数への呼び出し）

-wstv　Structured passed by value.（値によって渡された構造体）

-wuse　'XXXXXXXX' declared but never used.
（XXXXXXXX は宣言されているが1度も使われない）

■ Portability warnings（可搬性に関する警告）

-wapt *　Non-portable pointer assignment.
（可搬性がないポインタの代入）

-wcln　Constant is long.（定数が長すぎる）

-wcpt *　Non-portable pointer comparison.
（可搬性がないポインタの比較）

-wrng *　Constant out of range in comparison.
（比較において定数が指定の範囲を越えている）

-wrpt *　Non-portable return type conversion.
（可搬性がない戻り型の変換）

-wsig　Conversion may lose significant digits.
（変換により，有効数字が落とされる）

-wucp　Mixing pointers to signed and unsigned char.
（signed，unsigned char へのポインタを混ぜて使っている）

＊ はオンがデフォルト。他はすべてオフがデフォルト。

セグメント名の制御

-zA*name*　コードセグメントクラスの名前を ***name*** に変更します。デフォルトでは，コードセグメントにはクラス CODE が割りふられています。

-zB*name*　非初期化データセグメントクラスの名前を ***name*** に変更します。デフォルトでは，非初期化データセグメントにはクラス BSS が割りふられています。

-zC*name*　コードセグメントの名前を ***name*** に変更します。デフォルトでは，コードセグメントは_TEXT という名前がつけられます。ミディアム，ラージ，ヒュージの各メモリモデルは例外で，名前は ***filename*** _TEXT です（***filename*** はソースファイルの名前です）。

-zD*name*　非初期化データセグメントの名前を ***name*** に変更します。デフォルトでは，非初期化データセグメントは_BBS という名前がつけられます。ヒュージメモリモデルだけは例外で，非初期化データセグメントは生成されません。

-zG*name*　非初期化データセグメントグループの名前を ***name*** に変更します。デフォルトでは，データグループは DGROUP という名前がつけられます。ヒュージメモリモデルだけは例外で，データグループは存在しません。

-zP*name*　出力ファイルが，***name*** という名前のコードセグメントに対するコードグループとともに生成されるようにします。

-zR*name*　初期化データセグメントの名前に ***name*** をセットします。デフォルトでは，初期化データセグメントは_DATA という名前がつけられます。ヒュージモデルだけは例外で，セグメントは ***filename*** _DATA となります。

-zS*name*　初期化データセグメントの名前を ***name*** に変更します。デフォルトでは，データグループは DGROUP という名前がつけられます。ヒュージモデルだけは例外で，データグループは存在しません。

-zT*name* 初期化データセグメントクラスの名前に *name* をセットします。デフォルトでは，初期化データクラスセグメントはDATAという名前がつけられます。

-z*X* * *X* に対するデフォルト名を使います。たとえば，-zA*はコードセグメントに対するデフォルト名CODEを使います。

注意：8086のセグメント管理をよく理解している方以外は，これらのスイッチは使わないようにしてください。通常の状況では，セグメント名を指定する必要はほとんどありません。

コンパイル制御オプション

-B コンパイルを行ない，インラインアセンブリ言語コードを処理するためにアセンブラを呼び出します。

-c .Cファイルをコンパイル，.ASMファイルをアセンブルします。しかし，リンクは行ないません。

-o*filename* コンパイルを行ない，出力するオブジェクトファイル名を，*filename*.OBJにします。

-S ソースファイルをコンパイルし，アセンブリ言語出力ファイル(.ASM)を作ります。しかし，アセンブルは行ないません。

-E*filename* 使用するアセンブラプログラムの名前を *filename* にします。デフォルトではTASMが使用されます。

リンカオプション

-e*filename* 出力する実行可能プログラムのファイル名を *filename*.EXE にします。-e と *filename* の間に空白を入れてはならないことに注意してください。このオプションを指定がない場合,.EXE ファイルの名前には,最初に指定されたソースファイルのファイル名が使われます。

-M リンカに,フルリンクマップを作ることを指示します。デフォルトでは,リンクマップは作られません。

-l*x* オプション *x* をリンカに渡します。スイッチ-l-*x* はオプション *x* を抑制します。-l の後には複数のオプションを置くこともできます。リンカへ渡せるオプションについては付録 C の TLINK に関する節を参照してください。

環境オプション

-I*directory*

インクルードファイルを（標準の探索に加えて）探すディレクトリを指定します。*directory* には，ドライブ名およびサブディレクトリのパス名を指定します。ドライブ名は英字（大文字小文字どちらでもよい）1文字の後にコロン（：）をつけます。パス名にはどんなディレクトリパス名を指定してもかまいません。複数の-I オプションを指定することができます。

-L*directory*

リンカに，スタートアップオブジェクトファイル（C0*x*.OBJ）と Turbo C ライブラリファイル（C*x*.LIB，MATH*x*.LIB，EMU.LIB，FP87.LIB）を指定のディレクトリから取り込むことを指示します。デフォルトでは，リンカはカレントディレクトリでそれらのファイルを探索します。

-n*xxx* 出力する.OBJ や.ASM ファイルのディレクトリパス名を指定します。

Turbo C では，インクルードファイルとライブラリファイルについては，複数のディレクトリを探索することができます。つまり，インクルードディレクトリ（-I）およびライブラリディレクトリ（-L）オプションは，マクロ定義オプション（-D）と同様に，複数並べて書くことができます。

2つのオプションの書式は次の通りです。

ライブラリディレクトリ：	`-Ldirname[;dirname;...]`
インクルードディレクトリ：	`-Idirname[;dirname;...]`

dirname には，どのようなディレクトリパス名を指定してもかまいません。

複数のディレクトリは，次の3つの方法で指定することができます。

■1つの-I または-L オプションの後ろに，セミコロンで区切って複数のパス名を書く。

```
tcc -Ldirname1;dirname2;dirname3 -Iinc1;inc2;inc3 myfiile.c
```

■コマンドラインに複数の-I または-L オプションを置く。

```
tcc -Ldirname1 -Ldirname2 -Ldirname3 -Iinc1 -Iinc2 -Iinc3 myfiile.c
```

■上の2つを組み合わせる。

```
tcc -Ldirname1;dirname2 -Ldirname3 -Iinc1;inc2 -Iinc3 myfiile.c
```

複数の-I または-L オプションを並べた場合，コンパイラはそのすべてのディレクトリを左から右の順序で探索します。

注意：統合環境(TC.EXE)でも，上の最初の方法で複数のディレクトリを探索させることができます（**O**ptions/**D**irectories/**I**nclude directories, **O**ptions/**D**irectories/**L**ibrary directories)。

暗黙のライブラリとユーザ指定のライブラリ

Turbo C は，**暗黙のライブラリ**と**ユーザ指定のライブラリ**（**明示的なライブラリ**とも呼びます）という2種類のライブラリファイルを認識します。

■暗黙のライブラリファイルは，Turbo C が自動的にリンクするものです。これは，C*x*.LIB，EMU.LIB または FP87.LIB，MATH*x*.LIB，およびスタートアップファイル（CO*x*.OBJ）です。

■ユーザ指定のライブラリファイルは，コマンドライン上やプロジェクトファイル内で明示的に指定したものです。これは拡張子が.LIB のファイルです。

インクルードファイルとライブラリファイル探索のアルゴリズム

Turbo C のインクルードファイル探索のアルゴリズムでは，ソースコード内に並べられた# include ファイルは次のようにして探索されます。

■ソースコード内に# include <somefile.h>と書くと，Turbo C は指定されたインクルードディレクトリでのみ SOMEFILE.H を探します。

■これに対し，ソースコードで# include "somefile.h"と書くと，Turbo C はまずカレントディレクトリで SOMEFILE.H を探し，見つからなければ指定されているインクルードディレクトリを探します。

ライブラリファイル探索アルゴリズムもこれによく似ています。

■**暗黙のライブラリ**：Turbo C は指定されたライブラリディレクトリでのみ暗黙のライブラリを探索します。これは# include <somefile.h>と同様です。

■**明示的なライブラリ**：Turbo C が明示的なライブラリを探索する場所は，コマンドライン上のライブラリファイル名の記述方法によって異なります。

・ライブラリ名にドライブとディレクトリをつけない場合（たとえば mylib.lib など），Turbo C はまずカレントディレクトリを探し，そこになければ指定されているライブラリディレクトリを探します。これは，# include "somefile.h"の場合と同様です。

・ライブラリ名にドライブやディレクトリをつけた場合（c：mystuff¥mylib.lib など），Turbo C はライブラリパス名の一部として指定されたドライブとディレクトリだけを探し，指定されているライブラリディレクトリは探しません。

Turbo C バージョン2.0のライブラリ探索アルゴリズムは，従来のバージョンと上位コンパチです。

コンフィギュレーションファイル内での-L,-Iの使用

TCC.EXEとともに使われるTURBOC.CFG（コマンドラインのコンフィギュレーションファイル）の使い方がまだよく理解できていない場合は，第3章の「**TURBOC.CFGファイル**」を参照してください。

コマンドライン上に書かれた-Lおよび-Iオプションは，コンフィギュレーションファイル内に書かれたものより優先されます。これがどのように働くかは第3章の「**TURBOC.CFGファイル**」の節に述べられています。

ディレクトリの指定例

ここでは，複数のライブラリディレクトリ（-L）とインクルードディレクトリ（-I）オプションを含んだTCCコマンドライン例を示します。

1. カレントドライブがA：，カレントディレクトリはA：¥TURBOCで，そこにはTCC.EXEがあるとします。またBドライブのカレントポジションはB：¥ASTROLIBです。
2. インクルードファイル（.H，ヘッダファイル）はA：¥TURBOC¥INCLUDEにあります。
3. スタートアップファイル(C0T.OBJ, C0S.OBJ, . . . , C0H.OBJ)は，A：¥TURBOC¥STARTUPSにあります。
4. Turbo Cの標準ライブラリファイル（CS.LIB, CM.LIB, ..., MATHS.LIB, MATHM.LIB, ..., EMU.LIB, FP87.LIBなど）は，A：¥TURBOC¥LIBにあります。
5. “スター”システム用のユーザライブラリファイル（TLIBで作成・管理しているもの）は，A：¥TURBOC¥STARLIBにあります。その中の1つはPARX.LIBです。
6. “クェイサー”用のサードパーティ作成のライブラリは，B：¥ASTROLIBにあります。その中の1つはWARP.LIBです。

このような状況で，次のTCCコマンドラインを入力します。

```
tcc -mm -Llib;starlib -Iinclude orion umaj parx.lib b:¥astrolib¥warp.lib
```

TCC は ORION.C と UMAJ.C を.OBJ ファイルにコンパイルし，それらをミディアムモデルのスタートアップコード（C0M.OBJ），ミディアムモデルのライブラリ（CM.LIB, MATHM.LIB），標準の浮動小数点エミュレーションライブラリ（EMU.LIB），ユーザ指定のライブラリ（PARX.LIB, WARP.LIB）とリンクし，実行可能な ORION.EXE を作成します。

コンパイラは，ソースコード内で指定されているインクルードファイルを，A：¥TURBOC¥INCLUDE で探索します。

またコンパイラは，スタートアップコードを A：¥TURBOC に探しにいきます（そこに存在するので，探索は終了します）。さらに標準ライブラリを A：¥TURBOC¥LIB に探しにいきます（そこに存在するので探索は終了します）。

ユーザ指定のライブラリ PARX.LIB を探索するときには，コンパイラはまずカレントディレクトリ A：¥TURBOC を見ます。そこでは見つからないので，さらに次の順でライブラリディレクトリを探索します。まず A：¥TURBOC¥LIB，次に A：¥TURBOC¥STARLIB（ここに PARX.LIB があります）。

ライブラリ WARP.LIB については，明示的なパス（B：¥ASTROLIB¥WARP.LIB）が与えられているので，コンパイラはそこだけを探索します。

付録 C

Turbo C ユーティリティ

Turbo C のパッケージに入っているのは2種類の非常に高速な C コンパイラだけではありません。このパッケージには，強力なスタンドアロンのユーティリティが7つ含まれています。これらのユーティリティは，Turbo C のファイルやそれ以外のモジュールに対して使用するものです。

Turbo C に付属する7つのユーティリティは以下のとおりです。

■ CPP（Turbo C プリプロセッサ）
■ MAKE（スタンドアロンのプログラムマネージャ，TOUCH ユーティリティも含まれます）
■ TLINK（Turbo リンカ）
■ TLIB（Turbo ライブラリアン）
■ GREP（ファイル検索ユーティリティ）
■ BGIOBJ（ドライバ/フォント変換ユーティリティ）
■ OBJXREF（.OBJ ファイル・クロスリファレンスユーティリティ）

この付録では，プログラム例やコマンドラインの例をまじえて，各ユーティリティの内容と使用法について説明しています。

CPP:Turbo C プリプロセッサユーティリティ

CPP プリプロセッサは，インクルードファイルとマクロ定義を展開した C のソースプログラムのリストファイルを作成します。これは，C のプログラムを普通にコンパイルする際にはまったく必要ありません。

マクロやインクルードファイル内でコンパイルエラーが起こった場合，マクロ展開やインクルードファイル取り込んだ結果を見ることができれば，エラーの内容がもっとよくわかる場合がかなりあります。多くのコンパイラはマルチパス構成をとっているので，対応するパスの結果を調べることができます。

Turbo C は統合化された1パスのコンパイラなので，他のコンパイラの第1パスに相当するものとして CPP が用意されています。また，CPP をマクロプロセッサとして使用することもできます。

CPP は，スタンドアロンのコンパイラ TCC と同じようにして使用します。CPP は TCC と同じようにデフォルトのオプションを TURBOC.CFG から読み込みます。また，コマンドラインオプションも同じです。

TCC のオプションで CPP には関係ないものを指定した場合は，CPP はそのオプションを単に無視します。MS-DOS プロンプトに対して，cpp だけをタイプすると CPP のパラメータのリストを見ることができます。

1つの例外を除いて，CPP のコマンドラインで指定したファイル名は，TCC の場合と同様の扱いを受けます。ワイルドカードも使用できます。例外とは，指定したファイルはすべて C のソースファイルとして扱われるということです。.OBJ，.LIB，および.ASM ファイルに対して特別な取り扱いは行なわれません。

CPP で処理した各々のファイルに対して，結果はカレントディレクトリ（あるいは-n オプションで指定した出力ディレクトリ）に出力されます。出力されるファイルは，ソースファイルの拡張子を.I につけかえたものになります。

この出力ファイルは，ソースファイルとすべてのインクルードファイルの各行がふくまれるテキストファイルです。すべてのプリプロセッサ指令行は消去され，また条件コンパイルによってコンパイルの対象からはずされるテキスト行も削除されます。テキストの各行には，コマンドラインオプションでそうしない指定をしていなければ，その行のあったソースファイルのファイル名と行番号が先頭につけられます。また，テキスト行内のマクロは，すべて展開テキストに置き換えられます。

CPP の出力には各行の先頭にファイル名と行番号がつくので，これをコンパイルすることはできません。

マクロプロセッサとしてのCPP

-P オプションは，CPP に対して，各行にソースファイル名と行番号をつけるように指示します。しかし，-P-を指定すればCPP は行番号をつけないので，出力された.I ファイルは TC や TCC でコンパイルすることができ，CPP をマクロプロセッサとして使うことができます。

使用例

次に示す簡単な例で，-P および-P-それぞれの場合に，CPP がファイルをどのように処理するかを見てみましょう。

ソースファイル：HELLOJOE.C

```
/* This is an example of the output of CPP */
#define NAME "Joe Smith"
#define BEGIN {
#define END   }
main()
BEGIN
    printf("%s¥n",NAME);
END
```

CPP をプリプロセッサとして使用する場合のコマンドライン：

```
cpp hellojoe.c
```

出力：HELLOJOE.I

```
hellojoe.c 1:
hellojoe.c 2:
hellojoe.c 3:
hellojoe.c 4:
hellojoe.c 5:
hellojoe.c 6: main()
hellojoe.c 7: {
hellojoe.c 8: printf("%s¥n", "Joe Smith" );
hellojoe.c 9: }
```

CPP をマクロプロセッサとして使用する場合のコマンドライン：

```
cpp -P- hellojoe.c
```

出力：HELLOJOE.I

```
main()
{
printf("%s¥n", "Joe Smith" );
}
```

スタンドアロンの MAKE ユーティリティ

Turbo C の MAKE ユーティリティはインテリジェントなプログラムマネージャで，適切な命令が与えられていれば，プログラムの更新に必要な作業をすべてやってくれます。MAKE を実行すると，MAKE は以下のようなことを行ないます。

■ユーザが作った MAKEFILE を読み込みます。この MAKEFILE には，どの.OBJ ファイルおよびライブラリファイルをリンクして実行可能ファイル(.EXE)を作り出すのか，どのソースファイルおよびヘッダファイルをコンパイルして各.OBJ ファイルを作らねばならないのか，という MAKE への指示が書かれています。

■各.OBJ ファイルの日付と時刻を，その.OBJ ファイルが依存しているソースおよびヘッダファイルの日付・時刻と比較します。ソースあるいはヘッダファイルのいずれかが.OBJ ファイルよりも新しければ，MAKE は，そのファイルが変更されており，.OBJ ファイルを再コンパイルしなければならないことがわかります。

■ TCC を呼び出して，その.OBJ ファイルを再コンパイルします。

■すべての.OBJ ファイルの依存関係のチェックが済んだら，各.OBJ ファイルの日付・時刻を，実行可能ファイルの日付・時刻と比較します。

■.OBJ ファイルのどれかが.EXE ファイルより新しければ，Turbo リンカ TLINK を呼び出して，.EXE ファイルを再作成します。

実際には，MAKE には，プログラムを最新に保つために，これ以上のことを行なう能力があります。バックアップを作ったり，他のサブディレクトリからファイルを取り出したり，プログラムを自動的に走らせてデータファイルを更新したりすることもできます。MAKE を使えば使うほど，プログラム開発の助けとなる新しい使い方を見いだすことになるでしょう。

MAKE はスタンドアロンのユーティリティで，統合環境に含まれるプロジェクト Make とは別のものです。

この節では，TCC や TLINK に対してスタンドアロンの MAKE をどのように使うかを説明します。

簡単な例

では最初に，MAKEがいかに役立つかを示す例を見てみましょう。近くの恒星系に関する情報を画面に出力するプログラムを書いているとします。プログラムGETSTARSは恒星系をリストしたテキストファイルを読み込み，なんらかの処理をして，その結果をバイナリのデータファイルに出力します。

GETSTARSは，STARTDEFS.Hの中のいくつかの定義と，STARLIB.Cの中のいくつかのルーチン(STARLIB.Hで宣言されている)を使用します。また，プログラムGETSTARS自身も次の3つのファイルにわかれています。

- GSPARSE.C
- GSCOMP.C
- GETSTARS.C

最初の2つのファイル，GSPARSEとGSCOMPには，それぞれ対応するヘッダファイル(GSPARSE.H，GSCOMP.H)があります。3番目のファイルGETSTARS.Cにはプログラムの本体が含まれます。STARLIB内のルーチンを使うのは，3つのファイルの中でGSCOMP.CとGETSTARS.Cだけです。

各.Cファイルに必要なカスタムヘッダファイル(標準の実行時ライブラリルーチンを宣言しているTurbo Cヘッダ以外のもの）は次の通りです。

.Cファイル	カスタムヘッダファイル
STARLIB.C	なし
GSPARSE.C	STARDEFS.H
GSCOMP.C	STARDEFS.H, STARLIB.H
GETSTARS.C	STARDEFS.H, STARLIB.H, GSPARSE.H, GSCOMP.H

GETSTARS.EXEを生成するには（ミディアムデータモデルとします)，次のコマンド行を入力します。

```
tcc -c -mm -f starlib
tcc -c -mm -f gsparse
tcc -c -mm -f getcomp
tcc -c -mm -f getstars
tlink lib¥c0m starlib gsparse gscomp getstars,
      getstars, getstars, lib¥emu lib¥mathm lib¥cm
```

注意：MS-DOSでは，TLINKのコマンドは1行で入力しなければなりません。ここで2行にわけているのは，単にこのページの1行に入りきらないからです。

以上のことから，各ファイルには次のような依存関係があることがわかります。

■ GSPARSE，GSCOMP，GETSTARSは，すべてSTARDEFS.Hに依存しています。言いかえると，STARDEFS.Hに修正を加えたときは，この3つのファイルをすべて再コンパイルしなければなりません。
■同様に，STARLIB.Hに変更があると，GSCOMPとGETSTARSの再コンパイルが必要になります。
■ GSPARSE.Hを変更すると，GETSTARSを再コンパイルしなければなりません。GSCOMP.Hの場合も同じです。
■もちろん，ソースファイル(STARLIB.C，GSPARSE.Cなど)を修正したら，このファイルはコンパイルしなおす必要があります。
■最後に，再コンパイルを行なったときには，リンクをやり直さなければなりません。

憶えておくべきことがたくさんありますね。STARLIB.Hを修正したら，GETSTARS.Cをコンパイルすることになりますが，GSCOMP.Cの再コンパイルを忘れてしまうかもしれません。前述のコンパイル4回とリンク1回を行なうバッチファイルを作っておくこともできますが，それでは修正を行なうごとに何もかもやり直しになってしまいます。では，MAKEがこれをどのように簡単にしてくれるかを見てみましょう。

makefileを作る

makefileは，2種類のリストを組み合わせたテキストファイルです。2種類のリストというのは，ファイル間の依存関係とそれを満足するために必要なコマンドです。

例として，GETSTARSプログラムをのためのmakefileを作ってみましょう。これは次のようになるでしょう。

```
getstars.exe: getstars.obj gscomp.obj gsparse.obj starlib.obj
    tlink lib¥c0m starlib gsparse gscomp getstars, getstars, ¥
          getstars, lib¥emu lib¥mathm lib¥cm

getstars.obj: getstars.c stardefs.h starlib.h gscomp.h gsparse.h
    tcc -c -mm -f getstars.c

gscomp.obj: gscomp.c stardefs.h starlib.h
    tcc -c -mm -f gscomp.c

gsparse.obj: gsparse.c stardefs.h
    tcc -c -mm -f gsparse.c

starlib.obj: starlib.c
    tcc -c -mm -f starlib.c
```

これは，前のところで述べたことを別の形で表わしたものですが，並び方が逆になっている部分があります。MAKE はこのファイルを次のように解釈します。

- ファイル GETSTARS.EXE は，GETSTARS.OBJ，GSCOMP.OBJ，GSPARSE.OBJ，STARLIB.OBJ の4つのファイルに依存します。この4つのファイルのどれかに変更があった場合，GETSTARS.EXE は再リンクしなければなりません。これには，TLINK コマンドを使います。
- GETSTARS.OBJ は，GETSTARS.C，STARDEFS.H，STARLIB.H，GSCOMP.H の4つのファイルに依存します。この中のどれかのファイルに変更があった場合，与えられた TCC コマンドを使って再コンパイルをしなければなりません。
- GSCOMP.OBJ は，GSCOMP.C，STARDEFS.H，STARLIB.H の3つのファイルに依存します。この中のどれかのファイルに変更があった場合，与えられた TCC コマンドを使って再コンパイルをしなければなりません。
- GSPARSE.OBJ は，GSPARSE.C，STARDEFS.H の2つのファイルに依存します。この場合もどれかのファイルを変更したときは，与えられた TCC コマンドを使って再コンパイルをしなければなりません。
- STARLIB.OBJ は STARLIB.C だけに依存しています。STARLIB.C に変更があれば，TCC を使って再コンパイルしなければなりません。

これをどうするかというと，まず上のリストを入力して，MAKEFILE と名づけましょう。さあ，これで MAKE.EXE を使う準備はできました。

makefile を使う

先に述べた方法で MAKEFILE を作ったものとします。そしてもちろん，各種のソースファイルやヘッダファイルは存在しているものとします。後は次のコマンドを入力するだけです。

```
make
```

簡単ですね。MAKE は，MAKEFILE（別のファイル名もつけられますが，それについては後で説明します）を見つけて，GETSTARS.EXE の依存関係が書かれている1行目を読み込みます。MAKE は，GETSTARS.EXE が存在しているかどうか，また最新かどうかをチェックします。

GETSTARS.EXE が最新かどうかを明らかにするためには，GETSTARS.EXE が依存している各ファイル（GETSTARS.OBJ，GSCOMP.OBJ，GSPARSE.OBJ，STARLIB.OBJ）に対して同じことを調べてやる必要があります。今度は，各.OBJ ファイルがそれぞれ依存しているファイルについても調べなければいけません。.OBJ ファイルの更新に必要な TCC の呼び出しが行なわれ，最後に最新版の GETSTARS.EXE を生成するために，TLINK コマンドが（もし必要なら）実行されます。

もし，GETSTARS.EXE とすべての.OBJ ファイルがすでに存在していたらどうなるでしょう。その場合 MAKE は，各.OBJ ファイルが最後に更新された日時と，そのファイルの依存ファイルの日時を比較します。もし依存ファイルのどれかが新しければ，MAKE は.OBJ ファイルが作られた後で依存ファイルが修正されたと判断して，TCC コマンドを実行します。

.OBJ ファイルを更新していれば，その更新日時を GETSTARS.EXE と比較したときに，MAKE は，TLINK コマンドを実行して GETSTARS.EXE の最新版を生成しなければならないことがわかります。

途中経過

ここでは，今までの説明を明確にするために，前の例をステップごとに追っていきましょう。GETSTARS.EXE とすべての.OBJ ファイルが存在し，しかも GETSTARS.EXE はどの.OBJ ファイルよりも新しく，同様に各.OBJ ファイルはその依存ファイルのどれよりも新しいとします。この場合，

```
make
```

と入力しても何も起きません。更新の必要なものが何も無いからです。

では，STARLIB.C と STARLIB.H を修正した，たとえばなにかの定数の値を変更したとしましょう。今度は，

```
make
```

を入力すると，MAKE は，STARLIB.C が STARLIB.OBJ より新しいのを見て，次のコマンドを起動します。

```
tcc -c -mm -f starlib.c
```

次に，STARLIB.H が GSCOMP.OBJ より新しいのを見て，次のコマンドを起動します。

```
tcc -c -mm -f gscomp.c
```

また STARLIB.C は GETSTARS.OBJ よりも新しいので，次に起動するコマンドは，

```
tcc -c -mm -f getstars.c
```

となります。以上3つのコマンドによって，STARLIB.OBJ, GSCOMP.OBJ, GETSTARS.OBJ は，どれも GETSTARS.EXE よりも新しくなるので，MAKE は最後に次のコマンドを起動します。

```
tlink lib¥c0m starlib gsparse gscomp getstars,
    getstars, getstars, lib¥emu lib¥mathm lib¥cm
```

このコマンドは，すべてのファイルをリンクして新しいバージョンの GETSTARS.EXE を生成します（実際には TLINK のコマンド行は，1行に入っていないといけないことに注意してください）。

これで MAKE についての基本的な知識，MAKE とは何か，makefile をどう作るか，MAKE はファイルをどう解釈するか，を学んだことになります。今度は MAKE の詳細について見ていくことにしましょう。

makefileの作成

makefileには，MAKEがプログラムを最新に保つために必要な定義や相互関係が書かれています。makefileはいくつ作ってもよく，また好きな名前をつけることができます。MAKEFILEがデフォルトのファイル名で，MAKEを実行するときに特定のmakefileを指定しなければ，MAKEFILEが使われます。

makefileの作成には，Turbo Cエディタなど，ASCIIテキストエディタであればなにを使ってもかまいません。makefile内のルールや定義や命令はすべて改行で区切られます。1行が長すぎる場合には（たとえば前の例のTLINKコマンド），行末に円記号¥（IBM PCではバックスラッシュ＼）を置けば，次の行に続けることができます。

ホワイトスペース（空白とタブ）は，隣りあう名前（たとえば依存関係）を区切るため，およびルール中のコマンドにインデント（段下げ）をつける際に使用します。

makefileの要素

makefileを作るのは，定義あり，コマンドあり，命令ありで，ほとんどプログラムを書くようなものです。makefileに書くことのできる構成要素は次の通りです。

■コメント
■明示的ルール
■暗黙のルール
■マクロ定義
■命令：ファイルのインクルード，条件実行，エラー検出，マクロ定義の解除

では，各項目について詳しく見ていくことにしましょう。

コメント

コメントはシャープ記号（#）で始まります。MAKEはその行のうち，#以降の部分を無視します。コメントはどこにあってもかまいません。特定の桁位置から始めなければならないといった制限はありません。

円記号（¥）を使ってコメントを次の行に続けることはできません。各行に#をつける必要があります。さらに，コメントのある行は，¥で継続行にすることもできません。¥の

後ろに#があると，¥は行末ではなくなってしまいます。また¥が#の後ろにあると，¥もコメントの一部になってしまいます。

次に示すのは，makefile 中のコメントの一例です。

```
# GETSTARS.EXEを作成するmakefile
# 完全なプロジェクトの保守を行なう

getstars.exe: getstars.obj gscomp.obj gsparse.obj starlib.obj
# 次の行の最後にはコメントは置けない.
    tlink lib¥c0m starlib gsparse gscomp getstars, getstars, ¥
          getstars, lib¥emu lib¥mathm lib¥cm

# これは正しいコメント
getstars.obj: getstars.c stardefs.h starlib.h gscomp.h gsparse.h
    tcc -c -mm -f getstars.c    # ここにはコメントを書ける
```

明示的ルール

明示的ルールは，もうおなじみのはずです。これは今までの makefile の例で使ってきたものです。明示的ルールは次の形式をとります。

```
ターゲット [ターゲット...]: [ソース  ソース...]
    [コマンド]
    [コマンド]
      ...
```

ここで**ターゲット**は更新すべきファイル，**ソース**はターゲットが依存しているファイルで，**コマンド**は有効な DOS コマンドです（これには，.BAT ファイルの起動および.COM ファイルや.EXE ファイルの実行が含まれます)。

明示的ルールには1個以上のターゲット名，0個以上のソースファイルおよび実行すべきコマンドリストを任意個定義します。明示的ルールのターゲットやソースのファイル名のリストには，普通の MS-DOS のドライブ指定，ディレクトリ指定を書くことはできますが，ワイルドカードを使うことはできません。

次に示す文法が重要です。**ターゲットは行の先頭**（第1カラム）**になければなりません。**一方，各**コマンドは段下げしなければなりません**(先頭に少なくとも1つの空白またはタブを置く必要があります)。すでに述べたように，ソースファイルのリストやコマンドが1行に納まりきらない場合には継続文字として円記号（¥）を使うことができます。最後に，ソースのファイルとコマンドはともに省略することが可能です。**ターゲット[ターゲット...]**の後にコロンがついているだけの明示的ルールというのもありえます。

明示的ルールの基本的な考え方は，コマンド（あるいはコマンドリスト）によってソースファイルからターゲットが生成あるいは更新されるというものです。明示的ルールに出

会うと，MAKEはまずそのソースファイルが同じmakefile中のどこかでターゲットファイルになっていないかをチェックします。その場合には，まずそちらのルールが評価されます。

他の明示的ルール（あるいは暗黙のルール）に基づいてソースのファイルがすべて生成あるいは更新されたならば，MAKEはターゲットが存在するかどうかチェックします。存在しない場合には，指定された順番で各コマンドが起動されます。存在しているなら，そのファイルの更新日時を各ソースの更新日時と比較します。もしソースのどれかがターゲットより新しい日付で更新されていた場合は，コマンドのリストが実行されます。

MAKEの1回の実行で，1つのファイル名は明示的ルールの左辺には1度しか現われてはいけません。

明示的ルールの各コマンドは，ホワイトスペースで始まります。MAKEは次に1カラム目から始まる行（頭に空白のついていない行）かファイルの終わりが出てくるまで，明示的ルールの下の行はそのルールに対するコマンドだと解釈します。また，空白行は無視されます。

特別な場合

後ろにコマンドをまったく持たない明示的ルールは，コマンドのある明示的ルールとは少し違った扱いを受けます。

- ■あるターゲットに対する明示的ルールにコマンドが含まれる場合，そのターゲットが依存するファイルはその明示的ルール中で指定されたものだけです。
- ■明示的ルール中にコマンドのない場合，ターゲットは明示的ルールで指定されたファイルだけでなく，そのターゲットに対する暗黙のルールによって決定されるファイルにも依存します。

暗黙のルールについては後の節の説明を参照してください。

明示的ルールの例

次に明示的ルールの実例を示します。

```
myprog.obj: myprog.c
    tcc -c myprog.c

prog,.obj : prog2.c include¥stdio.h
    tcc -c -K prog2.c
```

```
prog.exe: myprog.c prog2.c include¥stdio.h
    tcc -c myprog.c
    tcc -c -K prog2.c
    tlink lib¥c0s myprog prog2, prog,, lib¥cs
```

■最初の明示的ルールは，MYPROG.OBJ が MYPROG.C に依存していること，MYPROG.OBJ は指定のコマンドを実行することにより生成されることを表わしています。

■同様に2番目のルールは，PROG2.OBJ が PROG2.C および（サブディレクトリINCLUDE 内の）STDIO.H に依存していること，このファイルは指定の TCC コマンドで生成されることを表わしています。

■最後のルールは，PROG.EXE が MYPROG.C, PROG2.C, STDIO.H に依存していること，この3つのどれかに変更があった場合には，与えられた一連のコマンドで PROG.EXE が更新できることを表わしています。しかし，これでは必要のない仕事までしてしまうことがあります。つまり，MYPROG.C だけを変更した場合でも PROG2.C も再コンパイルしてしまいます。これは，ルールのターゲットが最新ではないとわかった時点で，そのルールのコマンドがすべて実行されてしまうからです。

■ makefile の先頭に次のような明示的ルールを書き，

```
prog.exe: myprog.obj prog.obj
    tcc lib¥c0s myprog prog2, prog, , lib¥cs
```

その後に上述の（MYPROG.OBJ と PROG2.OBJ についての）2つのルールを並べれば，必要なファイルだけが再コンパイルされるようになります。

明示的ルールの場合には，C あるいはアセンブラソースファイルのインクルードファイルを追加したり削除したりしただけでも，いちいち makefile を書き直さなければなりません。MAKE で TC, TCC, および TASM を使うのであれば，この余分な作業を省略することができます（TC にコマンドラインオプション -b を使えば，MAKE から呼び出すことができます）。MAKE にコマンドラインオプション -a をつけることによって，自動依存チェックが行なわれます。

たとえば，次の makefile を考えてみましょう。

```
.c.obj:
    tcc -c $*
```

そして，foo.c という次のようなソースファイルを TCC（バージョン2.0以上）でコンパイルしたとします。

```
#include <stdio.h>
#include "dcl.h"

void  foo() {}
```

これでMAKEを次のコマンドラインで呼び出すと,

```
make  -a  foo.obj
```

foo.cだけでなく，stdio.hとdcl.hの日付・時刻もチェックされることになります。

暗黙のルール

MAKEでは**暗黙のルール**を定義することもできます。暗黙のルールは，明示的ルールを一般化したものです。それはどんなものなのでしょうか。

では実例を使って，この2つのルールの関係を見ていくことにしましょう。では，前の節の例題にある次の明示的ルールを考えてみましょう。

```
starlib.obj: starlib.c
    tcc -c -mm -f starlib.c
```

このルールは一般的なものです。なぜならこのルールは,.OBJファイルは同名の.Cファイルに依存していて,TCCの実行によって生成されるという一般的な原理に基づいているからです。実際，これと同じ書式の明示的ルールがいくつも（場合によっては何十も）並んでいるmakefileというのもありえます。

明示的ルールを暗黙のルールに書き直せば，同じ形式の明示的ルールを取り除くことができます。暗黙のルールは次のような形をしています。

```
.c.obj:
    tcc -c -mm -f $<
```

このルールの意味するのは次の通りです。

「.OBJで終わるファイルはすべて同じファイル名で.Cで終わるファイルに依存する。また，このファイルを生成するコマンドは

```
tcc -c -mm -f $<
```

である。ここで，$<はファイル名にソースの拡張子（.C）をつけたものを表わしている」。($<というシンボルはスペシャルマクロです。これについては次の節で説明しますが,これはコマンドが実行される時点で適切な.Cソースファイルの完全な名前に置き換えられるものです)。

暗黙のルールの構文を次に示します。

```
.ソース拡張子.ターゲット拡張子:
    [コマンド]
    [コマンド]
    ...
```

ここでも前と同様に，コマンドは省略可能です。また，段下げをする必要があります。

ソース拡張子は（1カラム目から始まらなければなりません）ソースファイルの拡張子です。つまり，このルールは次の形式のすべてのファイルに対応します。

```
ファイル名.ソース拡張子
```

同様に**ターゲット拡張子**は次のファイルを参照します。

```
ファイル名.ターゲット拡張子
```

ただし，**ファイル名**は両者で同じものです。言い方を変えれば，この暗黙のルールは任意のファイル名に対応する，次の書式の明示的ルールをすべて置き換えるものです。

```
ファイル名.ターゲット拡張子:  ファイル名.ソース拡張子
    [コマンド]
    [コマンド]
    ...
```

暗黙のルールが使われるのは，指定されたターゲットに対する明示的ルールが見つからない場合，および，そのターゲットに対する明示的ルールがコマンドを持たない場合です。

あるファイルに対してどの暗黙のルールが使用されるかは，そのファイルのファイル名の拡張子から決定されます。ファイル名のソース拡張子以外の部分が同じであるターゲットが存在する場合に，その暗黙のルールが使用されるわけです。

たとえば，次のような内容のmakefile（ファイル名はMAKEFILE）があるとします。

```
.c.obj
    tcc -c -ms -f $<
```

RATIO.CというCプログラムをRATIO.OBJにコンパイルする場合は，次のコマンドを使うことができます。

```
make ratio.obj
```

MAKEはRATIO.OBJをターゲットにしようとします。RATIO.OBJを生成するための明示的ルールはないので，MAKEは暗黙のルールを適用して次のコマンドを生成します。

```
tcc -c- ms -f ratio.c
```

このコマンドはもちろん，RATIO.OBJ の生成時に必要なコンパイルを行なうものです。暗黙のルールは，明示的ルールにコマンドが指定されていない場合にも使用されます。前述の例について，makefile の先頭に次の暗黙のルールがあるとしましょう。

```
.c.obj
    tcc -c -mm -f $<
```

すると，この後の明示的ルールは次のように書き直すことができます。

```
getstars.obj: stardef.h starlib.h gscomp.h gsparse.h
gscomp.obj: stardefs.h starlib.h
gsparse.obj: stardefs.h
```

ここでは，.OBJ ファイルをどうやって生成するかについて，明示的な情報を指定していませんので，MAKE はすでに定義されている暗黙のルールを使うことになります。また，STARLIB.OBJ は STARLIB.C だけに依存しているので，これについてのルールは，このリストからはずされています。MAKE が自動的に暗黙のルールを適用してくれます。

MAKE で自動依存チェックを有効にしている場合には，上の例のように.OBJ ファイルをターゲットとするすべてのルールを取り除くことができます。自動依存チェックを有効にして暗黙のルールを用いると，先程の MAKEFILE は次のようになります。

```
.c.obj:
    tcc -c -mm -f $<

getstars.exe: getstars.obj gscomp.obj gsparse.obj starlib.obj
    tlink lib¥c0m strlib gsparse gscomp getstars, getstars, ¥
          getstars, lib¥emu lib¥mathm lib¥cm
```

同じターゲット拡張子に対して複数の暗黙のルールを書くことも可能です。ただし，同時に適用できるのはその中の1つだけです。与えられたターゲット拡張子に対して2つ以上の暗黙のルールが存在する場合には，makefile にでてくる順番にルールのチェックが行なわれ，そのソース拡張子に一致するものが見つかるか，すべての適用可能なルールをチェックし終わるまでチェックを続けます。

MAKE は，最初にソース拡張子を持つファイルが見つかった暗黙のルールを使用します。そのルールのコマンドが実行できなかった場合でも，それ以上暗黙のルールのチェックは行ないません。

暗黙のルールの下に続く行は，先頭が空白でない行か，あるいはファイルの終わりに達するまで，そのルールに対するコマンドリストと見なされます。空白行は無視されます。コマンド行の構文については，この章の少し後の「**MAKE の使用**」の節で説明します。

特別な場合

明示的ルールと違い，暗黙のルールからではMAKEは完全なファイル名を知ることはできません。したがって，MAKEにはルールから得られるファイルの名前を扱うためのスペシャルマクロが用意されています（詳しくは少し後のマクロ定義についての説明を参照してください)。

暗黙のルールの例

次に暗黙のルールの例を説明します。

```
.c.obj:
    tcc -c $<

.asm.obj:
    tasm $* /mx;
```

最初の暗黙のルールの例では，ターゲットファイルは.OBJファイル，そのソースファイルは.Cファイルです。この例では，コマンドリストにはコマンド行が1行あります。コマンド行の構文については後にふれます。

2番目の例はMAKEに指定ファイルのアセンブルをさせようというものです。ソースは.ASMファイルで，TASMの/mxオプションを使用します。

コマンドリスト

ここまでで，明示的ルールと暗黙のルール，およびそれらのルールにコマンドリストがどう入るかについて述べました。それでは，こうしたコマンド自体，およびコマンド設定のオプションについて説明することにしましょう。コマンドリスト中のコマンドは段下げしなければなりません。つまり，先頭に少なくとも1個のブランクないしはタブが必要です。コマンドは次の形式をとります。

```
[ プレフィックス ...] コマンド本体
```

コマンドリスト内の各コマンドは，(省略可能な)プレフィックスのリストの後にコマンド本体が1つついたものです。

プレフィックス

プレフィックスを使って，MAKEがコマンドをどう扱うかそのコマンドの中から変更することができます。プレフィックスは，単価記号（@）か，マイナス記号（-）に数字を続けたもののどちらかです。

@　コマンドを実行する前に画面に表示するのを止めます。MAKEのコマンドラインで-sオプションを指定していない場合でも表示は行なわれません。このプレフィックスは，指定したコマンドに対してだけ適用されます。

-数字　終了コードをMAKEがどう扱うかを指定します。**数字**が指定されると，MAKEは終了コードが与えられた数字より大きいときだけ処理を中断します。次の例では，終了コードが4より大きいときにだけMAKEは異常終了します。

```
-4 myprog sample.x
```

-数字のプレフィックスが指定されていない場合は，MAKEはコマンドの終了ステータスをチェックして，0でない場合は処理を中断し，カレントのターゲットファイルを消去します。

-　-だけで数字のない場合には，MAKEは終了ステータスをまったくチェックしません。終了ステータスがどうであれ，MAKEは実行を続けます。

コマンド本体

コマンド本体は，COMMAND.COMに入力された場合とまったく同じように取り扱われます。ただし，リダイレクションとパイプはサポートされていません。次の組み込みコマンドを実行する際には，MAKEはコマンドを実行するためにCOMMAND.COMのコピーを起動します。

```
break    cd      chdir    cls     copy
ctty     date    del      dir     echo
erase    md      mkdir    path    prompt
rem      ren     rename   set     time
type     ver     verify   vol
```

これ以外のコマンド名に対しては，MAKEはMS-DOSの探索アルゴリズムを使ってコマンドを探します。

■まずカレントディレクトリを探します。次に PATH に指定されている各ディレクトリを探します。
■各ディレクトリ内では，まず拡張子.COM のファイルを探します。次に.EXE，最後に.BAT を探します。
■.BAT ファイルが見つかったときは，そのバッチファイルの実行のために COMMAND.COM のコピーが起動されます。

当然ながら，コマンドラインに拡張子が指定されている場合は，MAKE はその拡張子のファイルだけを探します。

コマンドの例

次のコマンドは，実行のために COMMAND.COM を起動します。

```
cd c:¥include
```

次のコマンドは，探索アルゴリズムをフルに使ってファイルを探します。

```
tlink lib¥c0s x y,z,z,lib¥cs
```

次のコマンドは，.COM ファイルのみを探します。

```
myprog.com geo.xyz
```

次のコマンドは，明示的にファイル名を与えて実行されます。

```
c:¥myprogs¥fill.exe -r
```

マクロ

makefile の中で，同じコマンドやファイル名，オプションなどを何度も繰り返し使うことがよくあります。この解説の最初の例題では，TCC コマンドはすべて-mm というスイッチを使っています。このスイッチはミディアムメモリモデルでコンパイルするという意味です。またこのときには，TLINK コマンドは C0M.OBJ，MATHM.LIB，CM.LIB というファイルを使用します。では，ここでラージメモリモデルを使用したい場合はどうなるでしょう。オプション-mm をすべて-ml に変え，さらに TLINK コマンドのファイルを適当なものに変更してやれば可能ですね。しかし，**マクロ**を定義するという手もあります。

マクロというのは，なんらかの文字列を表わす名前です。マクロ定義にはマクロ名と展開したテキストとを指定します。そうすれば，MAKE がマクロ名に出会ったときには，そ

のマクロ名を展開したテキストに置き換えます。

makefileの先頭で，次のマクロを定義したとしましょう。

```
MDL = m
```

こう定義することにより，マクロMDLは文字列mと同じものと見なされるようになります。そこで，makefileは次のように書き直してやることができます。

```
MDL = m

gerstars.exe: getstars.obj gscomp.obj gsparse.obj starlib.obj
    tlink lib¥c0$(MDL) starlib gsparse gscomp getstars,¥
          getstars, getstars, lib¥emu lib¥math$(MDL) lib¥c$(MDL)

getstars.obj: getstars.c stardefs.h starlib.h gscomp.h gsparse.h
    tcc -c -m$(MDL) getstars.c

gscomp.obj: gscomp.c stardefs.h starlib.h
    tcc -c -m$(MDL) gscomp.c

gsparse.obj: gsparse.c stardefs.h
    tcc -c -m$(MDL) gsparse.c

starlib.obj: starlib.c
    tcc -c -m$(MDL) starlib.c
```

メモリモデルを指定するところはすべて，マクロ呼び出し$(MDL)を使っています。MAKEの実行時には，$(MDL)は展開テキストのmに置き換えられます。その結果，前述の例と同じコマンドセットが得られます。

この修正で何が得られたのでしょう。それは柔軟性です。最初の行を，MDL=lに変更してやれば，コマンドはすべてラージメモリモデルを使用するためのものに変わります。さらに，1行目を取り去ると，MAKEを走らせる際に-D（定義）オプションを使って，次のようにどのメモリモデルを使うかを指定してやることもできます。

```
make -DMDL=l
```

ここではMAKEに，MDLを，展開テキストがlのマクロとして扱うように指示しています。

マクロの定義

マクロの定義は次の形式で行ないます。

```
マクロ名=展開テキスト
```

マクロ名はマクロの名前です。マクロ名は英字および数字からなる文字列で，ホワイトスペースを含んではなりません。ただし，マクロ名と等号（=）の間にはホワイトスペースを入れることができます。**展開テキスト**は任意の文字列で，行末で終了します。展開テキストは，行末に円記号を置いて次の行に続けることができます。

そのマクロ名がすでに定義されている場合には，makefile中のマクロ定義の場合でもMAKEのコマンド行の-Dオプションによるものであっても，新しい定義が元の定義に置き換わります。

マクロは大文字・小文字を区別します。つまり，マクロ名mdl, Mdl, MDLはすべて別々のものとして扱われます。

マクロの使用

マクロは，makefileの中で次の書式で呼び出されます。

$(マクロ名)

マクロ名は，たとえ1文字のものでも必らずカッコが必要です。ただし，後で説明する定義済みの特殊マクロは例外です。**$(マクロ名)**という構文は，**マクロ呼び出し**と呼ばれます。MAKEはマクロ呼び出しに出会うと，それを展開テキストに置き換えます。そのマクロが定義されていない場合には，空文字列に置き換えます。

特別な場合

マクロ内のマクロ：

マクロ定義の左辺（マクロ名）でマクロを呼び出すことはできません。マクロの使用が許されるのは右辺（展開テキスト）だけです。この場合，定義したマクロが呼び出されるまで展開は行なわれません。言い方を変えれば，マクロ呼び出しが展開されるときに，展開テキスト中に埋めこまれたマクロも展開されます。

ルール内のマクロ：

ルール行のマクロ呼び出しは，即座に展開されます。

命令内のマクロ：

!if命令および!elif命令中では，マクロ呼び出しは即座に展開されます。!if命令や!elif命令で呼び出そうとしたマクロがその時点で定義されていなかった場合は，値0（FALSE）に展開されます。

コマンド内のマクロ：

コマンド中のマクロ呼び出しは，そのコマンドの実行時に展開されます。

定義済みマクロ

MAKEには，$d, $*，$<，$：，$.，$&の6つの特殊マクロが組み込まれています。1つめは定義テストマクロで，条件実行命令!if および!elif 中で使用します。残りはファイル名マクロで，明示的ルールおよび暗黙のルールの中で使われます。さらに，その時点での SET の環境文字列がマクロとしてロードされます。また，マクロ__MAKE__は1と定義されています。

定義テストマクロ（$d）：

定義テストマクロ$dは，指定されたマクロ名が定義されているときには1に，されていないときには0に展開されます。そのマクロの展開テキストの内容には関係はありません。この特殊マクロは，!if 命令および!elif 命令の中でのみ使用できます。

たとえば，メモリモデルの指定のない場合には，ミディアムメモリモデルを使うようにmakefile を書き変えたいとしましょう。これは makefile の最初に次のように書けば可能です。

```
!if !$d(MDL)          # if MDL is not defined
MDL=m                 # define it to m (MEDIUM)
!endif
```

MAKE を起動するコマンド行が，

```
make -DMDL=1
```

ならば，MDL は 1 と定義されます。しかし，単に

```
make
```

で MAKE を起動した場合は，MDL は m と定義されます。これが“デフォルト”のメモリモデルというわけです。

各種のファイル名マクロは，同じような働きをします。つまり，生成しようとするファイルの完全パス名のバリエーションに展開されます。

ベースファイル名マクロ（$*）：

ベースファイル名マクロは，明示的ルールおよび暗黙のルールのコマンド内で使用し

ます。このマクロ（$＊）は，次のように生成すべきファイル名から拡張子を除いたものに展開されます。

```
ファイル名    A:¥P¥TESTFILE.C
$*の展開形    A:¥P¥TESTFILE
```

たとえば，前述のGETSTARS.EXEに対する明示的ルールは次のような形に書き直すことができます。

```
getstars.exe: getstars.obj gscomp.obj gsparse.obj starlib.obj
    tlink lib¥c0$(MDL) starlib gsparse gscomp $*, $*, $*, ¥
          lib¥emu lib¥math$(MDL) lib¥c$(MDL)
```

このルールのコマンドが実行される際には，マクロ$＊は（拡張子なしの）ターゲットのファイル名getstarsに置き換えられます。暗黙のルールの場合には，このマクロは非常に有用です。

たとえば，TCCを用いる暗黙のルールは次のような形をしています（マクロMDLの定義は他の場所で行なわれるとします。また，浮動小数点数ルーチンは使用しないものとしています）。

```
.c.obj:
    tcc -c -m$(MDL) $*
```

フルファイル名マクロ（$<）：

フルファイル名マクロも，明示的ルールおよび暗黙のルールのコマンド内で使用されます。明示的ルール中では，$<は次のように（拡張子も含む）ターゲットのファイル名全体に展開されます。

```
ファイル名    A:¥P¥TESTFILE.C
$<の展開形    A:¥P¥TESTFILE.C
```

たとえば次のルールは，

```
starlib.obj: starlib.c
    copy $< ¥oldobjs
    tcc -c $*
```

STARLIB.Cをコンパイルする前に，元のSTARLIB.OBJをディレクトリ¥OLDOBJSにコピーします。暗黙のルールの場合，$<はファイル名＋拡張子となります。たとえば，次のような前述の暗黙のルールは，

```
.c.obj:
    tcc -c $*.c
```

以下のように書き直すことができます。

```
.c.obj:
    tcc -c $<
```

ファイル名パスマクロ（$:）:

このマクロは次の例のように，（ファイル名の部分を除いた）パス名に展開されます。

```
ファイル名    A:¥P¥TESTFILE.C
$:の展開形    A:¥P¥
```

拡張子つきファイル名マクロ（$.）:

このマクロは次のように，拡張子も含めたファイル名に展開されます（ただしパス名は除かれます）。

```
ファイル名    A:¥P¥TESTFILE.C
$.の展開形    TESTFILE.C
```

ファイル名マクロ（$&）:

このマクロは次のように，パスや拡張子を除いたファイル名の部分だけに展開されます。

```
ファイル名    A:¥P¥TESTFILE.C
$&の展開形    TESTFILE
```

命令

Turbo C の MAKE の持つ**命令**の機能は，他の MAKE にはないものです。これは，C 自身のコンパイラ制御行と同様の機能です。こうした命令を使えば，他の makefile をインクルードしたり，ルールやコマンドを条件つきにしたり，エラーメッセージを出力したり，マクロを未定義に戻したりすることができます。

makefile 中の命令は，C の場合のシャープ記号（#）ではなく，行の先頭の感嘆符（!）で始まります。MAKE の命令には次のものがあります。

```
!include
!if
!else
!elif
!endif
!error
!undef
```

ファイルインクルード命令

ファイルインクルード命令(!include)は，指定したファイルをその命令を読み込んだ時点でmakefileに取り込みます。この命令の形式は次の通りです。

```
!include "ファイル名"
```

この命令は，任意の深さにネストすることができます。もし，インクルード命令がネストの外側ですでに取り込まれているファイルを取り込もうとした場合には，（したがって，ネストはループになるので）その内側のインクルード命令でエラーとなります。

この命令はどう使うのでしょう。では，次のような内容のファイルMODEL.MACを作成したとしましょう。

```
!if !$d(MDL)
MDL=m
!endif
```

こうすれば任意のmakefile中に，上の条件マクロ定義を，次の命令で取り込むことができます。

```
!include "MODEL.MAC"
```

!include命令に出会うと，MAKEは指定されたファイルをオープンし，その内容を読み込んでmakefile自体にあるように処理します。

条件実行命令

条件実行命令（!if, !elif, !else, !endif）を使用することによって，プログラマは柔軟なmakefileの作成が可能となります。ルールやマクロに条件を与えることができるので，（-Dを使った）コマンドラインでのマクロ定義でmakefileの一部を活かしたり殺したりすることができます。

これらの命令の書式は，Cのプリプロセッサのものと対応しています。

```
!if expression
[ lines ]
!endif

!if expression
[ lines ]
!else
[ lines ]
!endif

!if expression
[ lines ]
!elif expression
[ lines ]
!endif
```

注意：[lines]には次のものが許されます。

マクロ定義
明示的ルール
暗黙のルール
包含命令
if グループ
エラー命令
定義解除命令

条件命令はグループを形成します。グループには，少なくとも先頭の!if 命令と末尾の!endif 命令が必要です。

- ■グループ中には!else 命令を1つ含むことができます。
- ■!elif 命令は，!if 命令と!else 命令（あるいは!endif）の間に置くことができます。
- ■ルール，マクロ，条件命令以外の命令は条件命令の間にいくつでもはさむことができます。ただし，コマンドを持つルールを条件命令の両側にまたがるように書くことはできません。
- ■条件命令グループは任意の深さにネストすることができます。

個々のルール，コマンド，命令は同一のファイルに入っていなければなりません。

!if 命令は，必らず同一のソースファイル中に対応する!endif 命令が必要です。したがって，次のインクルードファイルは途中でどんなファイルを取り込んでいたとしても，対応する!endif 命令がないので誤りになります。

```
!if $(FILE_COUNT) > 5
    <なにかのルール>
!else
    <他のルール>
<ファイルの終わり>
```

条件実行命令中の式

式には1つの!if か!elif 命令を，Cのような文法で記述することができます。式は単に32ビットの符号つき整数として評価されます。数字は，8進，10進，16進定数として入力することができます。たとえば，次の式はすべて正しい定数です。

```
4536      #  10進定数
0677      #  8進定数
0x23aF    #  16進定数
```

式には次の単項演算子を使用できます。

-	負数
~	ビット反転
!	論理否定

式の中で使用できる二項演算子は次の通りです。

+	加算
-	減算
*	乗算
/	除算
%	剰余
>>	右シフト
<<	左シフト
&	ビットごとのAND
|	ビットごとのOR
^	ビットごとのXOR
&&	論理的AND
||	論理的OR
>	より大きい
<	より小さい

>=	等しいか大きい
<=	等しいか小さい
==	等しい
!=	等しくない

式の中で使用できる三項演算子は次の通りです。

? ： ?の前のオペランドは条件式として扱われます。
オペランドの値が0でない場合，2番目のオペランド(?と：にはさまれた部分)が結果となります。最初のオペランドの値が0の場合は，結果は3番目のオペランド（：の後の部分）の値となります。

式の中のオペランドをひとまとめにするためにカッコを使うことができます。カッコのない場合，二項演算子はC言語と同じ優先度でグループ化されます。

Cの場合と同様，同じ優先度の演算子の場合，グループ化は左から右へ行なわれます。ただし，三項演算子（? ：）の場合には右から左になります。

式の中でマクロ呼び出しをすることも可能です。特殊マクロ$d()も使用できます。マクロをすべて展開した後の状態で，式は正しい構文になっていなければなりません。展開後に式の構成要素でない単語が残った場合にはエラーになります。

エラー命令

エラー命令（!error）はMAKEの実行を止めて，!errorの後ろに指定したテキストを含む致命的エラーの診断メッセージを出力します。書式は次の通りです。

```
!error <任意のテキスト>
```

この命令は条件命令中に入れておいて，ユーザ指定による異常終了を可能にするために用意されています。たとえば，次のようなコードを最初の明示的ルールの前に入れておくことができます。

```
!if !$d(MDL)
# if MDL is not defined
!error MDL not defined
!endif
```

MDL を定義せずにこの部分が実行されると，MAKE は次のエラーメッセージを出して停止します。

```
Fatal makefile 5: Error directive: MDL not defined
```

定義解除命令

定義解除命令 (!undef) は，指定したマクロの定義を解除します。指定のマクロが未定義の場合は，この命令は何もしません。構文は次の通りです。

```
!undef マクロ名
```

MAKE の使用

ここまでで makefile について多くのことを学んできました。今度は makefile をどのように使うかについて見てみましょう。

コマンドラインの構文

MAKE を使用する最も簡単な方法は，MS-DOS のプロンプトに対して

```
make
```

と入力することです。この場合，MAKE はまず MAKEFILE というファイルを探します。もしこのファイルが見つからない場合，次に MAKEFILE.MAK を探します。このファイルも見つからなければ，エラーメッセージを出して停止します。

では，MAKEFILE や MAKEFILE.MAK 以外の名前のファイルを使いたい場合はどうすればいいでしょう。その場合は次のように MAKE にファイルオプション (-f) を指定します。

```
make -fstars.mak
```

MAKEの一般的な構文は次の通りです。

```
make オプション オプション ... ターゲット ターゲット ...
```

ただし，**オプション**はMAKEのオプション（後述），**ターゲット**は明示的ルールで扱うターゲットファイルの名前です。

この構文の文法は次の通りです。

■makeの後には空白を入れ，その後にMAKEのオプションを並べます。

■各オプションの間は空白で区切ります。オプションはどんな順序で並べてもかまいません。また（コマンドラインにおさまる限り）いくつ指定してもかまいません。

■オプションのリストの後には空白を入れ，その後にターゲットを並べます（ターゲットは指定しなくてもかまいません）。

■各ターゲットの間は空白で区切ります。MAKEはターゲットファイルを並んでいる順に評価して，必要とされる構成要素を再コンパイルします。

コマンド行にターゲットの指定のない場合には，MAKEは明示的ルールに最初に出てくるターゲットのファイルを使います。1行のコマンドで1つ以上のターゲットを指定した場合には，その内で必要なものが生成されます。以下にMAKEのコマンド行の例をもう少しあげておきます。

```
make -n -fstars.mak
make -s
make -Iinclude -DMDL=c
```

MAKEの中断について

MAKE中で実行されるコマンドをコントロールブレークで中断すると，MAKE自身も停止します。したがって，***CTRL-C***を押すと，実行中のコマンドおよびMAKEは停止します。

BUILTINS.MAKファイル

MAKEを使っていると，何度も使用するマクロやルール（通常は暗黙のルール）があることがわかります。そういったマクロやルールを扱うには3通りの方法があります。第1の方法は，作成するmakefile全部にいちいち書いていくやり方です。2番目は，こうしたマクロやルールを1つのファイルに入れておいて，各々のファイルで!include命令を使うとい

うものです。3番目の方法は，それらを BUILTINS.MAK というファイルに入れてしまうことです。

MAKE を実行すると，MAKE は BUILTINS.MAK というファイルを探しにいきます。このファイルが見つかると，MAKE は MAKEFILE（あるいは実行しようとする makefile）を処理する前に，まずこのファイルを読み込みます。

BUILTINS.MAK ファイルには，そのコンピュータ上で共通に使用されるルール（通常は暗黙のルール）やマクロを入れておくことができます。

BUILTINS.MAK は必らず必要なものではありません。BUILTINS.MAK があった場合には，MAKE はまずこのファイルを処理します。BUILTINS.MAK ファイルが見つからなかった場合は，すぐに MAKEFILE（あるいは指定した makefile）を処理します。

BUILTINS.MAK と makefile の探索

MAKE は，BUILTINS.MAK を，カレントディレクトリの中で探します。MS-DOS 3.x のもとであれば，次にスタートディレクトリ（MAKE.EXE が置かれているディレクトリ）を探します。したがって，(MS-DOS 3.*x*）を使っている場合には，BUILINS.MAK を MAKE.EXE と同じディレクトリに入れておくとよいでしょう。

MAKE は常に，makefile をカレントディレクトリでのみ探します。makefile には特定の実行可能プログラムを作成するためのルールが含まれています。この2つのファイルは同じ文法に従って記述します。

!include で指定したファイルもカレントディレクトリから探します。-I（インクルード）オプションを指定した場合は，そこで指定したディレクトリも探索の対象になります。

MAKE のコマンドラインオプション

MAKE のコマンド行オプションについては，すでにいくらか触れてきましたが，ここでオプション全部のリストを示します。オプションでは，大文字/小文字が区別されることに注意してください。たとえば，オプション-D のかわりに-d としてはいけません。

-a	自動依存チェックを行ないます。これは，統合環境のプロジェクト Make の自動依存チェックと同等の機能です。
-D 名前	指定の**名前**を1という文字1文字からなる文字列と定義します。
-D 名前＝文字列	指定の**名前**を等号の後の**文字列**と定義します。文字列には空白やタブを入れてはいけません。
-I ディレクトリ	MAKE はインクルードファイルを指定の**ディレクトリ**（およびカレ

	ントディレクトリ）から探します。
-U 名前	指定された**名前**の定義を解除します。
-s	通常 MAKE は各コマンドを実行する際に，そのコマンドを表示します。-s オプションを指定すると，実行される各コマンドを表示しなくなります。
-n	MAKE が，コマンドを表示するだけで実行はしないようにします。これは makefile のデバッグをする際に役立ちます。
-f ファイル名	指定した**ファイル名**を makefile として使用します。指定のファイルが存在せず，しかも拡張子を指定していない場合には，ファイル名.MAK というファイルを探します。
-? または -h	ヘルプメッセージを表示します。

MAKE のエラーメッセージ

MAKE の診断メッセージは，致命的エラーとエラーの2種類にわかれています。致命的エラーが起きた場合には，コンパイル作業は即座に中止されます。この場合，適当な措置を施してからコンパイルをやり直さなければなりません。一方，エラーとは makefile 中の文法的エラーや文脈上のエラーを示しています。この場合には，MAKE は makefile の処理を完了してから停止します。

致命的エラー

XXXXXXXX does not exist - don't know how to make

このメッセージが出力されるのは，ファイル生成のシーケンス中に存在していないファイルでしかもそのファイルを生成すべきルールのないものがあった場合です。

Error directive： XXXX

このメッセージは MAKE がソースファイル中の!error 命令を処理した場合に出力されます。命令中のテキストもメッセージ中に表示されます。

Incorrect command line argument： XXX

MAKE を実行する時のコマンド行に正しくないパラメタがあった場合には，このエラーが起きます。

Not enough memory

作業用の記憶領域が足りなくなった場合にのエラーが起きます。こういう場合には，もっと大きなメモリのマシンを使ってみてください。すでにメモリが640Kのマシンを使っている場合はソースファイルをもっと簡単にしてください。

Unable to execute command

このメッセージは，コマンドが実行できなかったときに出力されます。実行しようとしたコマンドが見つからなかった，あるいはコマンドのスペルをまちがえている場合などです。場合によっては，そのコマンドは見つかったけれども，内容が破損していたということもありえます。

Unable to open makefile

このメッセージは，カレントディレクトリにMAKEFILEまたはMAKEFILE.MAKというファイルがない場合に表示されます。

エラー

Bad file name format in include statement

インクルードするファイル名は二重引用符（"）で囲まなければいけません。ファイル名に先頭の二重引用符が抜けています。

Bad undef statement syntax

!undef 文は，文の本体には名前が1つだけで他には何も含んではいけません。

Character constant too long

文字定数は，1文字か2文字の長さしかとることができません。

Command arguments too long

MAKEが実行したコマンドの引数が，MS-DOSによる制限の127文字を越えています。

Command syntax error

このメッセージが出るのは次の場合です。

- ■ makefile の最初のルール行の先頭に空白があった。
- ■暗黙のルールが，. 拡張子. 拡張子：という形式でなかった。
- ■明示的ルールの：の前に名前の指定がなかった。
- ■マクロ定義の＝の前に名前がなかった。

Division by zero

!if 文中の除算あるいは剰余の計算で，0による除算が起こっています。

Expression syntax error in !if statement

!if 文中の式の形式が間違っています。カッコの対応がとれていない，演算子がたりないあるいは多すぎる，定数がたりないあるいは多すぎるといった場合です。

File name too long

!include 命令で指定したファイル名が長すぎてコンパイラが処理できません。MS-DOS のファイル名は64文字を越えてはいけません。

Illegal character in constant expression X

定数式内に許されない文字が含まれています。その文字が英字の場合はおそらく名前のスペルミスでしょう。

Illegal octal digit

8進定数の中に数字8あるいは9が含まれています。

Macro expansion too long

マクロを展開したものが4096文字を越えてはいけません。このエラーがよく起きるのは，マクロが再帰的に自分自身を展開している場合です。マクロは自分自身を展開することはできません。

Misplaced !elif statement

対応する!if 命令のない!elif 命令がありました。

Misplaced !else statement

対応する!if 命令のない!else 命令がありました。

Misplaced !endif statement

対応する!if 命令のない!endif 命令がありました。

No file name ending

インクルード文のファイル名の最後に二重引用符（"）がありません。

Redefinition of target XXXXXXXX

このファイル名は，2つ以上の明示的ルールの左辺に現われています。

Unable to open include file XXXXXXXX.XXX

指定のファイルが見つかりません。インクルードファイルが自分自身をインクルードしている場合にもこのメッセージが出力されます。指定のファイルが存在するかどうかチェックしてください。

Unexpected end of file in conditional started on line #

!endif なしにファイルが終わっています。!endif が抜けているか，スペルがまちがっています。

Unknown preprocessor statement

行の先頭に!がありますが，その後の文の名前が次のものではありません。

error　undef　if　else　include　else　endif

TOUCH ユーティリティ

ソースファイルに変更がなくても，ターゲットファイルのコンパイルやリンクをやり直したい場合があります。その方法の1つに，Turbo C に入っている TOUCH ユーティリティを使用するという手があります。TOUCH はファイルの更新日時を実行した時点に変更します。したがって，TOUCH をしたファイルは，そのファイルに依存しているファイルより新しくなります。

ターゲットファイルを作り直すには，そのターゲットが依存しているファイルのどれかに *TOUCH* します。ファイルに TOUCH するには，MS-DOS のプロンプトから次のように入力します。

```
touch ファイル名 [ファイル名 ... ]
```

すると，TOUCH はそのファイルの作成日時を更新します。

こうすれば，TOUCH したファイルに対応するターゲットファイルを作りなおすために MAKE を使うことができます（TOUCH では，MS-DOS のワイルドカード＊および?を使うことができます)。

TLINK：Turbo リンカ

Turbo C の統合開発環境 (TC) にはリンカが組み込まれています。コマンドライン版のTurbo C（TCC）では，リンカは別のプログラムとして起動します。Turbo C のリンカ，TLINK はスタンドアロンのリンカとしても使うことができます。

TLINK は小さくて軽いシステムです。他のリンカのような派手な機能は備えていませんが，非常に高速でコンパクトです。

デフォルトでは，TCC はコンパイルに成功すると TLINK を呼び出します。TLINK は，オブジェクトモジュールとライブラリファイルを組み合わせて実行可能なファイルを作成します。

ここでは，TLINK のスタンドアロンのリンカとしての使い方を説明します。

TLINK の起動

TLINK を起動するには，MS-DOS のコマンドラインで tlink とタイプし，必要に応じてパラメータをつけます。

パラメータなしで起動すると，次のようなパラメータとオプションの要約が表示されます。

```
Turbo Link  Version 2.0  Copyright (c) 1987, 1988 Borland International
Syntax:  TLINK objfiles, exefile, mapfile, libfiles
@xxxx indicates use response file xxxx
Options: /m = map file with publics
         /x = no map file at all
         /i = initialize all segments
         /l = include source line numbers
         /s = detailed map of segments
         /n = no default libraries
         /d = warn if duplicate symbols in libraries
         /c = lower case significant in symbols
         /3 = enable 32-bit processing
         /v = include full symbolic debug information
         /e = ignore Extended Dictionary
         /t = create COM file
```

この要約の2行目は次のようになっています。

```
Syntax:  TLINK objfiles, exefile, mapfile, libfiles
```

これは，ファイル名はこの順序で，種類の異なるファイルはカンマで区切って指定することを示しています。たとえば，コマンドラインで次のように指定したとします。

```
tlink /c mainline wd in tx,fin,mfin,lib¥comm lib¥support
```

TLINKはこのコマンドを，以下のように解釈します。

■リンクに際しては大文字/小文字を区別する（/c）。
■リンクする.OBJファイルは，MAINLINE.OBJ，WD.OBJ，LN.OBJ，およびTX.OBJである。
■実行可能プログラムの名前はFIN.EXEにする。
■マップファイルはMFIN.MAPにする。
■リンクするライブラリファイルは，COMM.LIBとSUPPORT.LIBで，ともにサブディレクトリLIBの中にある。

TLINKは，拡張子の指定のないファイルには，次のように拡張子を付加します。

■オブジェクトファイルには.OBJ
■実行可能ファイルには.EXE
■マップファイルには.MAP
■ライブラリファイルには.LIB

.EXEファイルの名前を指定しなかった場合，TLINKは先頭にあるオブジェクトファイルの拡張子を.EXEに置き換えたファイル名で実行可能ファイルを作成することに気をつけてください。たとえば上の例で，.EXEファイルの名前としてFINを指定しないと，MAINLINE.EXEという実行可能ファイルが作成されます。

/tオプションを使用したときには，実行可能ファイルの拡張子は，.EXEではなく.COMになることに注意してください。

TLINKは，コマンドラインで/xオプションを指定して，マップファイルを生成しないように指示しないかぎり，必らずマップファイルを作成します。

■/mオプションを与えると，マップファイルにパブリックシンボルが含まれます。
■/sオプションを与えると，マップファイルに詳細なセグメントマップが含まれます。

TLINKは，以下の規則に基づいてマップファイルの名前を決めます。

■.MAP ファイル名が指定されていない場合，実行可能ファイルの拡張子を.MAP に置き換えたものがマップファイル名になります(.EXE ファイル名の指定は，コマンドラインからでも，レスポンスファイルからでもかまいません。.EXE ファイル名の指定がなければ，先頭の.OBJ ファイル名によって決められます)。

■マップファイル名がコマンドライン（またはレスポンスファイル）で指定されている場合は，そのファイル名に拡張子.MAP を付加します。

なお，マップファイル名を指定した場合でも，/x オプションが指定されているとマップファイルは作成されません。

レスポンスファイルを使う

TLINK のさまざまなパラメータは，コマンドライン上でもレスポンスファイルからでも，また両方を組み合わせても指定することができます。

レスポンスファイルは，コマンドライン上では tlink の後に書き並べるオプションやファイル名が入っているテキストファイルです。

しかしコマンドラインとは違って，レスポンスファイルはテキストを何行でも続けることができます。行の最後にプラス記号（＋）をつければ，オブジェクトファイルやライブラリファイルの長いリストでも，複数の行にわけて書くことができます。

また，オブジェクトファイル，実行可能ファイル，マップファイル，ライブラリファイルの4つの部分を，それぞれ別の行に置くこともできます。ただしこの場合には，各部分の区切りにカンマを入れてはいけません。

最初に示したコマンドラインの例を，レスポンスファイル(FINRESP)に書きなおすと次のようになります。

```
/c mainline wd+
ln tx,fin+
mfin+
lib¥comm lib¥support
```

コマンドラインからは，次のようにして TLINK を呼び出します。

```
tlink @finresp
```

レスポンスファイルであることを示すために，ファイル名の前に単価記号（@）をつけなければならないことに注意してください。

さらに，リンクのコマンドを，複数のレスポンスファイルにわけて指定することもできます。たとえば，同じコマンドラインを次のような2つのレスポンスファイルにすることが

できます。

ファイル名	内容
LISTOBJS	mainline+ wd+ ln tx
LISTLIBS	lib¥comm+ lib¥support

このようにすると，TLINK コマンドは次のようになります。

```
tlink /c @listobjs,fin,mfin,@listlibs
```

Turbo C モジュールと TLINK

Turbo C は，6種類のメモリモデルをサポートしています(タイニィ，スモール，コンパクト，ミディアム，ラージ，ヒュージ)。TLINK を使って実行可能な Turbo C のファイルを作成する場合には，メモリモデルに応じた初期化モジュール（スタートアップファイル）とライブラリを取り込まなければなりません。

Turbo C のプログラムをリンクする際の，TLINK の一般的な形式は次のようになります。

```
tlink C0x <myobjs>, <exe>,[map],<mylibs> [emu|fp87 mathx] Cx
```

この中で指定するファイル名は，次のようになります。

〈*myobjs*〉	リンクしようとする.OBJ ファイル
〈*exe*〉	作成する実行可能ファイルの名前
[*map*]	マップファイルの名前（指定しなくてもよい）
〈*mylibs*〉	リンク時に取り込むライブラリファイル

これ以外のファイル名は，以下のような Turbo C のファイルを示します（*x* は，メモリモデルを表わす英字1文字で，*t*, *s*, *c*, *m*, *l*, *h* のいずれかです）。

C0*x*	*x* メモリモデル用の初期化モジュール
emu ｜ fp87	浮動小数点ライブラリ（どちらか1つを選ぶ）
math*x*	*x* メモリモデル用の数学ライブラリ
Cx	*x* メモリモデル用の実行時ライブラリ

注意：タイニィモデルを使用して，TLINK に.COM ファイルを作成させたい場合は，/t オプションを指定してください。

初期化モジュール

初期化モジュールは C0*x*.OBJ という形式の名前で，*x* はメモリモデルを表わすアルファベット1文字（*t*, *s*, *c*, *m*, *l*, *h*）です。適切な初期化モジュールがリンクされないと，識別名が未解決であるとか，スタックが生成されないといったエラーメッセージの長いリストが出力されます。なお，初期化モジュールは通常スタートアップファイルと呼ばれています。

また初期化モジュールは，オブジェクトファイルのリストの先頭に指定しなければなりません。初期化モジュールは，プログラムのさまざまなセグメントの順序を決めているので，これが最初にないと各セグメントが正しく配置されないため，わけのわからないバグが生じてしまいます。

TLINK のコマンドラインでは，.EXE ファイルの名前をはっきりと指定するようにしてください。そうしないと実行可能ファイルの名前は，（おそらく意図していない）C0*x*.EXE になってしまいます。

ライブラリ

コマンドラインにはさらに，ユーザ独自のライブラリの後に，メモリモデルに応じたライブラリを置かなければなりません。こうしたライブラリは，一定の順序で並べる必要があります。まず浮動小数点ライブラリと適切な数学ライブラリ（この2つは省略することもできます），次にメモリモデルに対応する実行時ライブラリです。この順にそって，ライブラリについて説明していきましょう。

Turbo C のグラフィックス関数を使用している場合は，GRAPHICS.LIB をリンクしなければなりません。このグラフィックスライブラリは，どのメモリモデルにも共通です。

Turbo C のプログラムの中で浮動小数点を使っている場合は，浮動小数点ライブラリ（EMU.LIB または FP87.LIB）と数学ライブラリ（MATH*x*.LIB）をリンクしなければな

りません。

2つの浮動小数点ライブラリは，どのメモリモデルにも共通です。

■浮動小数点のエミュレーションロジックを使うと，数値演算コプロセッサ（8087, 80287）のあるマシンでもないマシンでも，プログラムを実行することができます。この場合には EMU.LIB を使う必要があります。

■数値演算コプロセッサのあるマシンでしか実行しないプログラムの場合には，FP87.LIB を使用すれば，実行可能プログラムはよりコンパクトにかつ高速になります。

数学ライブラリは MATH*x*.LIB という名前で，*x* はメモリモデルを表わすアルファベット1文字（*s*, *c*, *m*, *l*, *h*）です（タイニィモデルは，スモールと同じ MATHS.LIB を使用します）。

TLINK のコマンドラインで，常にエミュレーションライブラリと数学ライブラリを指定してもかまいません。指定があっても，プログラムの中で浮動小数点を扱っていなければ，これらのライブラリからは何も取り込まれません。ただし，浮動小数点を使っていないとわかっている場合には，これらのライブラリを指定しないでおけば，リンクの時間を少し短縮することができます。

C の実行時ライブラリは，プログラムのメモリモデルに対応したものを，いつも必らずリンクしなければなりません。実行時ライブラリは C*x*.LIB という名前で，*x* はメモリモデルを表わすアルファベット1文字（*s*, *c*, *m*, *l*, *h*）です。

注意：浮動小数点演算を使用している場合には，浮動小数点ライブラリと数学ライブラリを，実行時ライブラリより前に指定しなければなりません。そうしないとリンクが行なえません。

TCC から TLINK を使う

スタンドアロンの Turbo C コンパイラ TCC を，TLINK のフロントエンドとして使うことができます。この場合，TCC が適当なスタートアップファイル，ライブラリ，実行可能ファイル名を指定して TLINK を起動します。

このためには，TCC のコマンドラインで，.OBJ, .LIB などの拡張子も含めて，正確なファイル名を指定します。たとえば，次のコマンドラインを入力すると，

```
tcc -mx mainfile.obj sub1.obj mylib.lib
```

TCC は，TLINK を呼び出して，C0*x*.OBJ，EMU.LIB，MATH*x*.LIB，C*x*.LIB（*x* メモ

リモデル用の，初期化モジュール，デフォルトの8087エミュレーションライブラリ，数学ライブラリ，実行時ライブラリ）というファイル名を渡します。TLINKは，以上のファイルと，ユーザのモジュールMAINLINE，OBJとSUB1.OBJ，およびライブラリMYLIB.LIBとをリンクします。

注意：TCCがTLINKを呼び出す際には，（-l-cによって無効にされていない限り）必らず/cオプション（大文字/小文字を区別する）が使用されます。

TLINKのオプション

TLINKのオプションは，コマンドライン上のどこに置いてもかまいません。オプションは，スラッシュ（/）の後にオプションを示す文字（*m,x,i,l,s,n,d,c,3,v,e,t*）を書きます。

複数のオプションを指定する場合，オプションの間に空白を入れる必要はありません（/m/cと/m　/cは同じです）。また，コマンドライン上でとびとびに置いてもかまいません。以下では，各オプションを説明していきます。

/x, /m, /s オプション

デフォルトでは，TLINKは必らず実行可能ファイルのマップを作成します。このデフォルトのマップに含まれるのは，プログラム内のセグメント名のリスト，プログラムのスタートアドレス，リンク中に起きた警告およびエラーメッセージだけです。

より詳細なリンクマップが必要であれば，/mオプションを指定して，アドレス順にソートされたパブリックシンボルのリストを，マップファイルに追加することができます。このようなマップファイルはデバッグの際に有効です。SYMDEBなど多くのデバッガは，パブリックシンボルのリストを使って，シンボルでアドレスが参照できるようになっています。

/sオプションを指定すると，/mオプションで作成されるのと同じセグメント，パブリックシンボル，スタートアドレスの他に，詳細なセグメントマップが追加されます。以下に示したのは，詳細なセグメントマップの例です。

アドレス	長さ(バイト)	クラス	セグメント名	グループ	モジュール	アラインメントおよび結合
0000:0000	0E5B	C=CODE	S=SYMB_TEXT	G=(none)	M=SYMB.C	ACBP=28
00E5:000B	2735	C=CODE	S=QUAL_TEXT	G=(none)	M=QUAL.C	ACBP=28
0359:0000	002B	C=CODE	S=SCOPY_TEXT	G=(none)	M=SCOPY	ACBP=28
035B:000B	003A	C=CODE	S=LRSH_TEXT	G=(none)	M=LRSH	ACBP=20
035F:0005	0083	C=CODE	S=PADA_TEXT	G=(none)	M=PADA	ACBP=20
0367:0008	005B	C=CODE	S=PADD_TEXT	G=(none)	M=PADD	ACBP=20
036D:0003	0025	C=CODE	S=PSBP_TEXT	G=(none)	M=PSBP	ACBP=20
036F:0008	05CE	C=CODE	S=BRK_TEXT	G=(none)	M=BRK	ACBP=28
03CC:0006	066F	C=CODE	S=FLOAT_TEXT	G=(none)	M=FLOAT	ACBP=20
0433:0006	000B	C=DATA	S=_DATA	G=DGROUP	M=SYMB.C	ACBP=48
0433:0012	00D3	C=DATA	S=_DATA	G=DGROUP	M=QUAL.C	ACBP=48
0433:00E6	000E	C=DATA	S=_DATA	G=DGROUP	M=BRK	ACBP=48
0442:0004	0004	C=BSS	S=_BSS	G=DGROUP	M=SYMB.C	ACBP=48
0442:0008	0002	C=BSS	S=_BSS	G=DGROUP	M=QUAL.C	ACBP=48
0442:000A	000E	C=BSS	S=_BSS	G=DGROUP	M=BRK	ACBP=48

このマップには，各モジュールの各セグメントごとに，アドレス，長さ(バイト数)，クラス，セグメント名，グループ，モジュール，および ACBP 情報が含まれます。

同一のセグメントが複数のモジュールにある場合には，1つのモジュールが複数の行に現れます（上の例でいえば SYMB.C)。セグメントマップの内容は，ACBP フィールド以外は特に説明は必要ないでしょう。

ACBP フィールドは，インテルの定義にしたがって，A (alignment)，C (combine)，および B (big) のアトリビュートを4ビットフィールドのセットにエンコードしたものです。TLINK が使用するのは，この中で A，C，B の3つのフィールドだけです。マップの中では，ACPB の値は16進数で表わされます。この値は，次に示すような各フィールドの値の OR をとったものです。

フィールド	値	意味
Aフィールド	00	絶対セグメント
(アラインメント)	20	バイトアラインメントのセグメント
	40	ワードアラインメントのセグメント
	60	パラグラフアラインメントのセグメント
	80	ページアラインメントのセグメント
	A0	名前のない絶対メモリ領域
Cフィールド	00	結合不可
(コンビネーション)	08	パブリックコンバインセグメント
Bフィールド	00	64未満のセグメント
(ビッグ)	08	64Kちょうどのセグメント

/l オプション

/l オプションを指定すると，.MAP ファイルの中にソースコードの行番号に対応するセクションが生成されます。このオプションを使用するためには，コンパイルの際に-y (**O/C/C/Line numbers...On**) を指定して.OBJ ファイルを生成しておく必要があります。TLINK にマップファイルを作成しないように指定した場合 (/x オプション)，/l オプションは無視されます。

/i オプション

/i オプションを指定すると，データレコードを持たないセグメントに対しても，実行可能ファイルの中に初期化されていないセグメントを付け加えます。通常このオプションを使用する必要はありません。

/n オプション

/n オプションはリンカに，ある種のコンパイラがオブジェクトファイルの中に生成するデフォルトライブラリ情報を無視するように指示します。TLINK はデフォルトライブラリのサーチをサポートしていないので，デフォルトライブラリが別のディレクトリに置かれている場合には，このオプションが必要になります。他の言語で書かれたモジュールリンクするようなときには，このオプションを使う場合が出てくるでしょう。

/c オプション

/c オプションを指定すると，パブリックおよび外部シンボルに関して大文字と小文字を区別するようになります。デフォルトでは，たとえば fred，Fred，FRED はすべて同じものとみなされます。/c オプションが指定されていると，これらはそれぞれ別のものとして認識されます。

/d オプション

通常 TLINK は，同一のシンボルが複数のライブラリファイルの中にあっても，警告を出しません。プログラムにそのようなシンボルを取り込む場合，TLINK はコマンドラインで先に指定されているファイルにあるシンボルのコピーを使用します。これは一般的な機能であり，TLINK はシンボルの重複に対してデフォルトでは警告を出さないようになっています。以下では，ある状況を想定して，この機能をどのように使うのかについて説明します。

ライブラリが2つあるとします。1つは SUPPORT.LIB，もう1つは補助的なライブラリ DEBUGSUP.LIB です。この2つのライブラリに，重複しているルーチンがいくつかあるとしましょう（ただし，DEBUGSUP.LIB の中のルーチンは，SUPPORT.LIB のものとは少し機能が異なるもの，たとえばデバッグ用のバージョンだと考えてください）。リンクのコマンドで，DEBUGSUP.LIB を先に指定すると，実行可能ファイルに取り込まれるのは，SUPPORT.LIB のものではなく，DEBUGSUP.LIB 中のルーチンになります。

この機能を使いたくない場合，あるいはどのルーチンが重複しているのかはっきりわからない場合には，/d オプションを指定することができます。これを指定すると，TLINK は，ライブラリの中で重複しているルーチンを，プログラムでは使用しないものも含めてすべてリストアップします。

/d オプションを指定すると，さらに，.OBJ ファイルと.LIB ファイルに同じシンボルが

ある場合にも警告を出します。この場合にもやはり，コマンドライン上で先に指定した(左側に置いた)ファイル中のシンボルがリンクされるので，使用されるシンボルは.OBJ ファイル内のものになります。

Turbo C に付属しているライブラリ内のシンボルは，通常どんなリンクコマンドに対しても重複することはありません。EMU.LIB と FP87.LIB (あるいは CS.LIB と CL.LIB) には確かに重複シンボルがありますが，これらが一回のリンクで同時に使用されることはありません。またたとえば，EMU.LIB，MATHS.LIB，および CS.LIB の中には重複するシンボルはありません。

/e オプション

Turbo C のディスクに含まれているすべてのライブラリファイルには，TLINK のリンクを高速化するための情報からなる拡張辞書が含まれています。拡張辞書は，TLIB に対して/E オプションを指定すれば，どんなライブラリファイルにも付け加えることができます(この後の TLIB の節を参照してください)。

拡張辞書を持つライブラリのリンクは確かに速いのですが，拡張辞書を使わないことを指示する/e スイッチを使いたくなる理由が2つあります。

■拡張辞書を使うと，リンクの際にメモリが少し余分に必要になります。
■/e を指定しないと，TLINK は拡張辞書を持つライブラリに含まれているデバッグに関する情報を無視してしまいます。

/t オプション

タイニィモデルでコンパイルしたオブジェクトファイルを/t スイッチをつけてリンクすると，TLINK は，通常の.EXE ファイルではなく.COM ファイルを生成します。

/t が使用された場合，実行可能ファイルのデフォルトの拡張子は.COM になります。

注意：.COM ファイルは，サイズは64K を越えられず，セグメント相対フィックスアップを持つことはできず，また開始アドレスは0：100H でなければなりません。実行可能ファイルの拡張子として.COM 以外を使用する場合(たとえば.BIN など)，スタートアドレスは0：0か0：100H のいずれかです。

/v オプション

/v オプションは，TLINK に実行可能ファイルの中にデバッグ情報を組み込むように指示するためのものです。

/3 オプション

/3オプションは，いくつかのオブジェクトモジュールが，TASM あるいはその互換アセンブラによって生成されたもので，かつ80386プロセッサ用の32ビットコードを含んでいる場合のために使用するものです。このオプションは TLINK が必要とするメモリを増やし，またリンクのスピードを下げます。したがって，本当に必要な場合にのみ指定してください。

制限事項

先に述べたように，TLINK はコンパクトなリンカであり，過剰と思われるオプションは持っていません。TLINK の制限事項で重要と思われるものを以下に示します。

- ■オーバーレイはサポートしていません。
- ■コモン変数は部分的にしかサポートしていません。コモン変数を解決するにはパブリックが必要です。
- ■論理セグメントの数は最大で約4000です。
- ■名前とクラスが同一のセグメントは，すべて結合されるか，すべて結合されないかのどちらかです（これはアセンブラプログラマにとっては障害と考えられるかもしれません)。
- ■ Microsft C，または Microsoft FORTRAN でコンパイルしたコードは，TLINK でリンクすることはできません。これは，Microsoft 言語の.OBJ ファイルには，情報が公開されていないオブジェクトレコードフォーマットが使用されており，TLINK はこれをサポートしていないためです。

TLINK は，Turbo C(統合環境とコマンドライン版の両方)に加えて，TASM や Turbo Prolog，およびその他のコンパイラとともに使用することを目的に設計されています。しかし，MS-LINK と全面的に置き換えられるものではありません。

エラーメッセージ

TLINKには，致命的なエラー，致命的でないエラー，警告，という3種類のエラーがあります。

- 致命的なエラーが起きると，TLINKはただちに処理を中止し，.EXEファイルは削除されます。
- 致命的でないエラーが起きた場合には，.EXEファイルと.MAPファイルは削除されませんが，この.EXEファイルは実行しないようにしてください。
- 警告はその名のとおり，修正したほうがよいと考えられる状態についての報告です。警告が出ても，.EXEファイルと.MAPファイルは生成されます。

この節で述べているエラーメッセージには，次のような名前と値が示されます。実際のエラーメッセージでは，これらは適切な名前および値に置き換えられます。

<sname>	シンボル名
<mname>	モジュール名
<fname>	ファイル名
<lsegname>	論理セグメント名
XXXXh	4桁の16進数の後に'h'がつく

致命的なエラー

致命的なエラーがあると，TLINKは処理を中止して.EXEファイルを削除します。

XXXXXXXX.XXX： bad object file

オブジェクトファイルが壊れています。このエラーは多くの場合，指定されたオブジェクトファイルのコンパイルが完全には行なわれていないことが原因です。コンパイル中にシステムをリセットしたり，***CTRL-STOP***でコンパイルを中断した場合などにディスク上に作られている，不完全なオブジェクトファイルをリンクしようとしていることが考えられます。

XXXXXXXX.XXX： unable to open file

指定されたファイルが存在しない（あるいはタイプミスをした）場合に起きます。

Bad character in parameters

コマンドラインまたはレスポンスファイルに，次のいずれかの文字

" * < = > ? [ ] |

あるいはタブ，復帰，改行，***CTRL-Z*** 以外の制御文字がありました。

msdos error, ax = xxxxh

これは，MS-DOS コールで予期しないエラーが返された場合に起こります。表示される *ax* の値はエラーコードです。これは，TLINK の内部エラーあるいは MS-DOS のエラーであることを示しています。TLINK が使用する MS-DOS コールでこのエラーが起こり得るのは，リード，ライト，シークおよびクローズです。

Not enough memory

リンクを行なうための充分なメモリがありません。メモリに常駐させているアプリケーションがあればそれをとりはずすなど，メモリのユーザエリアを大きくする必要があります。

〈lsegname〉: segment/group exceeds 64K

別々のソースファイルにある同じ名前のセグメントを結合する際に，1つのデータ(またはコード）セグメントで定義されるデータが多すぎる場合に，このエラーが出力されます。また，あるグループのセグメントを結合する際に，そのグループが64K を越えてしまう場合にも，このエラーが出ます。

Undefined symbol name

コールされている関数 *name* は，カレントファイル，およびリンクされる他のモジュールあるいはライブラリのいずれにも存在していません。シンボル名は，定義された関数名と正確に一致していなければなりません。

Invalid group definition

通常このエラーは，コンパイラの生成したコードが正しくない場合にしか起こりません。Turbo C のオブジェクトファイルでこのエラーが出た場合は，再コンパイルを行なってみてください。それでも同じ結果であれば，MSA カスタマーサポートセンターに連絡してください。

Invalid segment definition

通常このエラーは，コンパイラの生成したコードが正しくない場合にしか起こりません。Turbo C のオブジェクトファイルでこのエラーが出た場合は，再コンパイルを行なってみてください。それでも同じ結果であれば，MSA カスタマーサポートセンターに連絡してください。

Unknown option

コマンドラインまたはレスポンスファイルにスラッシュがありましたが，その次の文字はオプションとして認められません。

Write failed, disk full?

TLINK がディスクに書き込もうとしたデータが全部書けなかったときにこのエラーが出ます。ほとんどはディスクがいっぱいになっているのが原因です。

Relocation table full

リンクされるファイルに，標準の DOS のリロケーションテーブルが保持できる数を越えるベースフィックスアップがあります（ベースフィックスアップは多くの場合 far 関数の呼び出しによって生成されます）。

32-bit record encountered in modele XXXX： use "/3" option

このエラーは，/3オプションが指定されていないときに，オブジェクトファイルに（80386の）特殊な32ビットレコードが含まれていたときに起こります。/3オプションをつけて TLINK をもう一度実行してください。

Invalid entry point offset

このエラーは，32ビットレコードがリンクされるときにのみ発生します。プログラムの初期エントリポイントのオフセットが，DOS の制限である64K を越えていることを意味しています。

Invalid initial stack offset

このエラーは，32ビットレコードがリンクされるときにのみ発生します。スタックポインタの初期値が，DOS の制限である64K を越えていることを意味しています。

Base fixup offset overflow

このエラーは，32ビットレコードがリンクされるときにのみ発生します。ベースフィックスアップのオフセットが，DOS の制限である64K を越えていることを意味しています。

Cannot generate COM file： invalid initial entry point address

/t オプションが指定されていますが，プログラムの開始アドレスが，.COM において必要な100H ではありません。

Cannot generate COM file： segment-relocatble items present

/t オプションが指定されていますが，プログラムに，.COM では許されないセグメント相対フィックスアップが含まれています。

Cannot generate COM file： program exceeds 64K

/t オプションが指定されていますが，プログラムサイズのトータルが.COM ファイルの制限である64K を越えてしまいます。

Cannot generate COM file： stack segment present

/t オプションが指定されていますが，プログラムで，.COM ファイルでは許されないスタックセグメントを宣言しています。

致命的でないエラー

TLINK の致命的でないエラーは2つだけです。すでに述べたように，致命的でないエラーが起きても.EXE ファイルと.MAP ファイルは削除されません。しかし統合環境では，この2つのエラーは致命的なエラーと同様に扱われます。

XXX is unresolved in module YYY

モジュールの中で参照されているシンボルが，指定されたどのオブジェクトファイルあるいはリブファイルでも定義されていません。シンボルのスペルが正しいかどうか確認してください。Turbo C モジュールのシンボルに対してこのエラーが起きている場合，異なるファイルにあるシンボルの宣言で，pascal と cdecl などの型が一致していない，あるいはプログラムが必要とする.OBJ ファイルを指定し忘れていることなどが考えられます。

Fixup overflow in module XXXXXXXX, at 〈lsegname〉： xxxxh, target = 〈sname〉

これは，オブジェクトファイルの中で，TLINK が解決しなければならないデータあるいはコードへの参照が正しく行なわれていないことを示しています。

このメッセージは，主にメモリモデルが一致していない場合に出力されます。原因としては，別のコードセグメントにある関数を near で呼び出していることなどが考えられます。また，データ変数や関数を参照するデータを near で呼び出している場合にもこのエラーが出ます。どちらの場合においても，エラーメッセージの中の *target* の値が，参照されている変数名あるいは関数名です。この参照はエラーメッセージで示されているモジュール内で行なわれているので，そのモジュールのソースファイルにおける参照をチェックしてください。

この方法でエラーの原因が見つからない場合，あるいはアセンブラや Turbo C 以外の高級言語でプログラミングしている場合には，別の原因も考えられます。Turbo C の場合でも，セグメント名またはグループ名に，使用するメモリモデルのデフォルト以外のものを指定していると，このエラーが出ることがあります。

警告

TLINK の警告は3つだけです。2つは，シンボル定義の重複をに関するもの，もう1つはタイニィモデルに適用されるもので，スタックが定義されていないことを知らせます。

Warning： XXX is duplicated in module YYY

シンボルが，1つのモジュールの中で2回定義されています。これは Turbo C のオブジェクトファイルでも起こることがあります。文字ケース（大文字/小文字）が異なる2つの pascal 名が定義されている場合などです。

Warning： XXX defined in module YYY is duplicated un module ZZZ

シンボルが2つのモジュールで重複して定義されています。これは，コマンドラインで同じオブジェクトファイルを2回指定している場合や，重複するシンボルの一方はスペルミスである場合などが考えられます。

Warning： no stack

このメッセージは，リンクされるどのオブジェクトファイルあるいはリブファイルでも，スタックセグメントが定義されていない場合に出力されます。これは，Turbo C のタイニィメモリモデルなど，.COM ファイルに変換されるプログラムに対しては，正常な状態で出力されるメッセージです。

Turbo C のタイニィ以外のメモリモデルに対してこのメッセージが出力された場合は，スタートアップファイル（C0*x*.OBJ）に正しいものを指定したかどうか確認してください。

TLIB：Turbo ライブラリアン

TLIBは，ボーランド社のTurboライブラリアンです。これは，個々のオブジェクトモジュール（.OBJファイル）のライブラリを管理するユーティリティです。ライブラリは，オブジェクトモジュールの集合をひとまとまりとして扱う大変便利な方法です。

Turbo Cに含まれているライブラリはTLIBで作成されています。TLIBを使うと，ユーザ独自のライブラリを作ることができ，またTurbo Cライブラリ，ユーザライブラリ，他のプログラマが作成したライブラリ，市販のライブラリなどに変更を加えることができます。TLIBは，次のような場合に用いることができます。

- ■オブジェクトモジュールの集合から新しいライブラリを**作成する。**
- ■既存のライブラリへ，オブジェクトモジュールや他のライブラリを**追加する。**
- ■既存のライブラリから，オブジェクトモジュールを**削除する。**
- ■既存のライブラリ中のオブジェクトモジュールを**置き換える。**
- ■既存のライブラリからオブジェクトモジュールを**取り出す。**
- ■新規作成した，または既存のライブラリの**内容を表示する。**

既存のライブラリに変更を加えた場合，TLIBは必らず元のライブラリのコピーに.BAKという拡張子をつけて保存しておきます。

TLIBは，リンクを高速化させるために使用される拡張辞書を作成する（あるいはライブラリファイルの中に含める）こともできます。これについては，/Eオプションの節を参照してください。

TLIBは，Turbo Cで実行可能なプログラムを作成する際に必須というわけではありませんが，プログラム開発には便利なツールです。大規模なプログラム開発でのTLIBの必要性は，いずれおわかりになるでしょう。他の人が開発したオブジェクトモジュールライブラリを利用する場合にも，必要に応じてTLIBでそれを管理していくことができます。

オブジェクトモジュールライブラリを使用する利点

Cのプログラミングでは，実行時ライブラリの関数のような，役に立つ関数をまとめて作ることがよくあります。Cは分割コンパイルができるので，こうした関数は別々にコンパイルするソースファイルにわけておくことが多いでしょう。あるプログラムでは，これらた

くさんの関数のうちの一部しか使わない場合に，使用するファイルをいちいち指定するのはとても面倒なことです。一方，いつもすべてのソースファイルをインクルードすると，プログラムが巨大になり，きわめて扱いにくくなってしまいます。

オブジェクトモジュールライブラリを使えば，Cの関数の集まりを管理する際の問題が解決されます。プログラムをライブラリとリンクする場合，リンカはライブラリの中を調べて，そのプログラムに必要なモジュールだけを選び出してくれます。またライブラリにしておけば，.OBJファイルを別々に置いておくよりも（特に各オブジェクトファイルが小さい場合には），小さなディスク空間ですみます。さらに，ライブラリを使うと，リンクの速度を上げることにもなります。というのはオブジェクトモジュールすべてをオープンするのではなく，1つのファイルをオープンするだけですむからです。

TLIBコマンドラインの要素

TLIBを起動するには，MS-DOSのプロンプトに対し，TLIBコマンドラインを入力します。tlibだけをタイプして**リターン**を押すと，TLIBの使い方の要約が表示されます。

TLIBコマンドラインは次のような一般形をしています。カッコ[ ]で囲んだ項目は省略できる場合があります。

```
tlib ライブラリ名 [/c] [操作指定] [リストファイル]
```

ここではコマンドラインの各構成要素について説明し，そのあとTLIBの使い方について詳しく述べます。最後にTLIBの使用例を示します。

構成要素	説明
tlib	TLIB を起動するコマンド名です。
ライブラリ名	作成または管理を行なうライブラリの MS-DOS におけるパス名です。ワイルドカードは使用できません。拡張子を指定しない場合は.LIB とみなされます。TCC および TC のプロジェクト Make では，ライブラリファイルの識別に拡張子.LIB を必要とするので，これ以外の拡張子はおすすめしません。 指定したライブラリが存在せず，操作指定が add の場合は，ライブラリを新たに作成します。
/C	大文字/小文字区別のフラグです。このオプションは通常使われることはありません。詳しい説明は，「**高度な操作：/C オプション**」を参照してください。
/E	拡張辞書を作成します。詳しくは「**拡張辞書を作成する：/E オプション**」の解説を参照してください。
操作指定	TLIB が実行する操作の指定です。操作指定はどんな順番で行なってもかまいません。単にライブラリの内容を調べたい場合は，操作指定を与える必要はありません。
リストファイル	ライブラリの内容のリスト（一覧）を出力するファイル名です。リストファイルを指定する場合は，必らずその前にカンマを置かなければなりません。ファイル名を指定しないとリストは生成されません。リストには，各モジュールが ABC 順で表示されます。各モジュールには，そのモジュールで定義されているグローバルシンボルが ABC 順で示されます。リストファイル名のデフォルトの拡張子は.LST です。 リストを画面に表示するにはリストファイル名を CON にし，プリンタに出力するには PRN という名前を使います。

操作指定リスト

操作指定リストには，TLIBに実行させる仕事を指定します。複数の操作指定を並べることができます。**操作指定**は，1文字あるいは2文字の動作シンボルの後に，ファイル名あるいはモジュール名を続けたものです。動作シンボル，ファイル名あるいはモジュール名の前後には空白を置いてもかまいませんが，2文字の動作シンボルの間，およびファイル名，モジュール名の途中に置くことはできません。

コマンドラインには，127文字(MS-DOSのコマンドラインの制限)を越えなければ，操作をいくつ並べてもかまいません。操作指定の順序は重要ではなく，TLIBは操作を以下のような決まった順序で実行します。

1. 最初に，**抽出**操作（extract）を行ないます。
2. 次に，**削除**操作（remove）を行ないます。
3. 最後に，**追加**操作（add）を行ないます。

モジュールの置き換えは，まず削除(remove)した後，かわりのモジュールを追加(add)して行ないます。

ファイル名とモジュール名

TLIBがオブジェクトモジュールファイルをライブラリに追加する場合，そのファイル名は単に**モジュール**と呼ばれます。TLIBは，与えられたファイル名からドライブ名とディレクトリ名および拡張子を取り去り，モジュール名とします。

TLIBは，指定がなければ適当なデフォルト値をとることに注意してください。たとえば，カレントディレクトリにある拡張子.OBJのついたモジュールを指定する場合，モジュール名だけを与えればよく，パスや拡張子.OBJを指定する必要はありません。

ファイル名あるいはモジュール名には，ワイルドカードは使用できません。

TLIBの操作

TLIBは3種類の**動作シンボル**(-, +，＊)を認識します。これらは単独で，あるいは2つ組み合わせて，合計5種類の異なる操作が可能となります。2文字の操作では，文字の順番は重要ではありません。次に，動作シンボルとその動きについて説明を行ないます。

動作シンボル	名前	説明
+	追加	指定したファイルをライブラリに追加します。ファイル名に拡張子がついていなければ，.OBJ とみなします。ファイルそれ自身がライブラリ（拡張子.LIB）の場合は，その中のすべてのモジュールを対象ライブラリに追加します。 追加されるモジュールがすでに存在している場合は，メッセージを出力し，新しいモジュールの追加は行ないません。
−	削除	指定したモジュールをライブラリから削除します。ライブラリ中に指定のモジュールがない場合は，メッセージを表示します。
*	抽出	ライブラリから，指定したファイル名に対応するモジュールをコピーして，ファイルを作成します。モジュールが存在しない場合は，メッセージを表示し，ファイルは作成されません。指定したファイルがすでに存在する場合には上書きされます。
−+ +−	置換	指定したモジュールを，対応するファイルで置き換えます。これは，削除に続いて追加を行なう操作指定の短縮形となります。
−* *−	抽出削除	指定したモジュールを対応するファイルにコピーし，そのモジュールをライブラリから削除します。これは抽出に続いて削除を行なう操作指定の短縮形となります。

削除では，モジュール名のみ必要とするわけですが，ドライブと拡張子を含むフルパス名を入力してもかまいません。モジュール名以外はすべて無視されます。

ライブラリ内のモジュール名を変更することはできません。これを行なうには，まず抽出削除を行なって，作成されたファイル名を変更し，最後にそれを追加します。

ライブラリの作成

ライブラリを作成するには，ディスク上にないライブラリにモジュールを追加すること

になります。

レスポンスファイルの利用

たくさんの操作指定を行なう場合，あるいは決まりきった操作を何度も繰り返さなければならない場合などは，**レスポンスファイル**を利用すると便利です。レスポンスファイルは，TLIB コマンドが書き込まれている通常のテキストファイルで，Turbo C エディタで簡単に作成することができます。レスポンスファイルを使うと，コマンドライン1行には入りきらないような TLIB コマンドも実行することができます。

レスポンスファイルを使うには，レスポンスファイル名を TLIB コマンドラインの好きなところに指定します。

- ■レスポンスファイルは，複数行のテキストで構成することができます。各行の終わりには&を置きます。&は，次の行が継続行であることを表わします。
- ■ TLIB コマンド全体をレスポンスファイルに入れる必要はなく，コマンドの一部を入れ，残りはコマンドラインで入力することも可能です。
- ■ TLIB のコマンドラインには，複数のレスポンスファイルを指定することができます。

この節の最後の「**TLIB の使用例**」に，レスポンスファイルとそれを使用する TLIB コマンドラインの例があります。

拡張辞書を作成する：/E オプション

大きなライブラリファイル(たとえば標準の C*x*.LIB ライブラリなど)のリンクを高速化させるために，TLIB に，拡張辞書を作成してライブラリファイルにそれをつけ加えるように指示することができます。この辞書には，標準の辞書には含まれていない情報が非常にコンパクトな形で入っています。この情報があると，TLINK はライブラリファイルをより速く処理できます。ディスクがフロッピィディスクあるいは遅いハードディスクの場合には特にそうです。Turbo C のマスターディスクの中に入っているライブラリには，すべて拡張辞書が含まれています。

修正中のライブラリに対して拡張辞書を作成するには，ライブラリに対してモジュールを追加，削除，置換するために TLIB を呼び出す際に，単に/E オプションを指定します。

既存のライブラリで，修正はしたくないが拡張辞書は作成したいという場合は，/E オプションを使って，TLIB にそのライブラリに存在しないようなモジュールを削除するよう指示します。TLIB は「指定されたモジュールはそのライブラリには存在しない」という警告を出しますが，そのライブラリに対して拡張辞書を作ってくれます。たとえば，次のように入力します。

```
tlib /E mylib -bogus
```

高度な操作：/C オプション

モジュールをライブラリに追加する際には，TLIB はライブラリ内の各モジュールで定義されているグローバルシンボルのテーブルを管理しなければなりません。ライブラリ内のすべてのシンボルは，互いに異なるものでなければなりません。シンボルの二重登録になるようなモジュールをライブラリに追加しようとすると，TLIB がメッセージを表示し，モジュールの追加は行なわれません。

通常 TLIB がライブラリ内のシンボルの二重登録をチェックする際は，大文字と小文字の区別は行ないません。たとえば，シンボル lookup と LOOKUP は重複するものとみなします。C 言語は大文字と小文字を区別する言語なので，すでにライブラリ内にあるシンボルと，文字ケース（大文字小文字の違い）のみが異なるシンボルを含むモジュールを追加する場合には，/c オプションをつける必要があります。/c オプションは，すでにライブラリ内にあるシンボルと，文字ケースが異なるシンボルを含むモジュールを受け入れることになります。

C は大文字と小文字の区別をする言語であるのに，/c オプションを指定しないと，文字ケースのみが異なるシンボルを受け入れないのは，少し変に思えるかも知れません。これは，ライブラリ中の文字ケースのみが異なるシンボルを区別しないリンカがあるからです。TLINK を使う限りは，/c オプションを使用しても何の問題も引き起こしません。

しかし，ライブラリを他のリンカで利用したい（あるいは，他の人が他のリンカで，そのライブラリを利用したい）と思っている場合には，/c オプションは使うべきではないでしょう。

TLIB の使用例

次に簡単な例をいくつか示します。

1. X.OBJ, Y.OBJ, Z.OBJ を含む MYLIB.LIB という名のライブラリを作成するには，次のように入力します。

```
tlib mylib +x +y +z
```

2. 1.と同じ操作で，さらにリストファイル MYLIB.LST をとりたい場合は，次のように入力します。

```
tlib mylib +x +y +z, mylib.lst
```

3. 既存の CS.LIB というライブラリのリスト CS.LST をとるには，次のように入力します。

```
tlib cs, cs.lst
```

4. MYLIB.LIB に対して，モジュール X.OBJ を新しいものと置き換え，A.OBJ を追加し，Z.OBJ を削除するには，次のように入力します。

```
tlib mylib -+x +a -z
```

5. MYLIB.LIB からモジュール Y.OBJ を抽出し，さらにリスト MYLIB.LST をとるには，次のように入力します。

```
tlib mylib *y, mylib.lst
```

6. レスポンスファイルを使用して，モジュール A.OBJ, B.OBJ, . . . , G.OBJ からなる新しいライブラリ ALPHA.LIB を作成するには，まず次のような内容のテキストファイル ALPHA.RSP を作成します。

```
+a.obj +b.obj +c.obj &
+d.obj +e.obj +f.obj &
+g.obj
```

そして，リストファイル ALPHA.LST をとる TLIB コマンドを次のように与えます。

```
tlib alpha @alpha.rsp, alpha.lst
```

GREP：ファイル検索ユーティリティ

GREPは強力な検索ユーティリティで，一回で複数のファイルの中からテキスト（文字列）を探し出すことができます。

GREPの一般的なコマンドラインの書式は次の通りです。

```
grep [オプション] 検索文字列 ファイル指定 [ファイル指定...]
```

たとえば，setupmodemというテキストを含んでいるソースファイルを見つけたい場合には，次のようなコマンドラインでGREPを呼び出します。

```
grep setupmodem *.c
```

カレントディレクトリ内のすべての.Cファイルの内容から，setupmodemという文字列を探すことができます。

GREPのオプション

コマンドラインで指定する**オプション**は，ダッシュ（-）の後ろに1つあるいは複数の文字を並べたものです。各々の文字は，オンまたはオフを指定できるスイッチです。プラス（+）を文字の後につけるとそのオプションはオンになり，ダッシュ（-）を文字の後につけるとそのオプションはオフになります。

デフォルトはオンとなっています(+を意味します)。たとえば，-rは-r+と同じことを意味します。複数のオプションをわけて書くことも（-i -d -lなど），まとめてかくことも（-ildあるいは-il -dなど）できます。これらはすべて同じオプションを表わします。

GREPで用いるオプション文字とその意味を示します。

-c　**行数のみ**：マッチ（一致）した行数だけを表示します。少なくとも一行はマッチした行を含むファイルについては，ファイル名とマッチした行の数を表示します。マッチした行そのものは表示しません。

-d　**ディレクトリ**：ファイル指定にパスが含まれている場合，そのディレクトリの中，およびそのディレクトリの下のすべてのサブディレクトリの中にある，ファイル指定にマッチするすべてのファイルについて検索します。ファイル指定にパスが含まれていない場合，ファイルはカレントディレクトリにあるものとみなします。

-i **文字ケースの無視**：大文字小文字の違いを無視します。a～zの文字を，それぞれA～Zの対応する文字と同じであるとみなします。

-l **ファイル名の表示**：マッチした行を含むファイルの名前のみを表示します。マッチする行を1行見つけると，ファイル名を表示した後，直ちに次のファイル中の検索に移ります。

-n **行番号**：マッチした各行を，先頭に行番号をつけて表示します。

-o **UNIX出力フォーマット**：マッチした行の出力フォーマットを変えて，UNIX風のコマンドラインパイプを容易にサポートできるようにします。出力される行の前には，その行を含んでいる行のファイルの名前が付加されます。

-r **正規表現による検索**：検索文字列で指定したテキストを，そのままの文字列ではなく，正規表現として扱います。

-u **アップデートオプション**：GREPはコマンドラインで与えられたオプションとデフォルトのオプションを合わせてGREP.COMの中に書き込み，これを新たなデフォルト値とします（言い変えれば，GREPは自分自身を変更するのです）。このオプションにより，自分の好みに合ったデフォルト値の設定を作り上げることができます。

-v **マッチしない行**：マッチしない行だけを表示します。検索する文字列を含まない行を，マッチしない行と考えます。

-w **ワードサーチ**：正規表現に一致すると認識されたテキストは，その前後の文字がワード（一単語）を構成する文字ではない場合にのみ，「一致する」とされることになっています。ワードを構成するデフォルトの文字はA～Z，9～0，それに下線（_）です。このオプションには他の形式もあり，それを使うとワードを構成する文字セットを指定することができます。その形式は-w[*set*]というもので，*set*は正規表現による文字集合の定義です。英字が入っている場合は，自動的にその文字に対応する大文字，小文字が定義に含められます。たとえ探索が大文字小文字の区別をして行われる場合でもそうです。-wオプションが-uオプションと組み合わせて使われた場合，そのワード構成文字集合はデフォルトの文字集合となります。

-z **詳細表示**：検索するファイルごとに（マッチする行がなくても）ファイル名を表示します。また，マッチした行は先頭に行番号をつけて表示されます。さらに，ファイルごとにマッチした行の数を（たとえそれが0でも）表示します。

優先順位

各オプションはスイッチとなっていることを思い出してください。最後に切り換えた状

態が有効になっているのです。各オプションはオンかオフかのいずれかだけです。コマンドライン上で与えたオプションの指定は，同じオプションの以前の状態を無効にします。たとえば，次のようなコマンドラインを入力したとしましょう。

```
grep -r -i- -d -i -r- main( my*.c
```

このコマンドラインを与えると，-d オプションがオン，-i オプションがオン，-r オプションがオフの状態で動作します。

-u オプションを用いると，各オプションのデフォルトの設定を，好みに応じてGREP.COM に組み込むことができます。たとえば，常に詳細表示(-z がオン)を行なうようにしたければ，次のようなコマンドでこれを組み込みます。

```
grep -u -z
```

探索する文字列

探索文字列の値は，GREP が探索するパターンを決定します。探索する文字列は，**正規表現**，あるいは**リテラル文字列**のいずれかです。正規表現では，いくつかの文字は特別な意味を持っており，探索を制御する演算子として機能します。リテラル文字列の場合は，そのような演算子はありません。各文字は，文字そのものとして扱われます。

空白（スペース）やタブを区切り記号とみなされるのを防ぐのに，探索する文字列を引用符で囲むことができます。行をまたがっての探索はおこなわれません。

正規表現は，1文字，あるいは大カッコ[]で囲まれた複数の文字からなります。いくつかの正規表現を連結したものも正規表現となります。

正規表現の中の演算子

-r オプションを用いると，探索する文字列は（リテラル文字列ではなく）正規表現とみなされます。この中で，以下の文字は特別な意味を持っています。

^	正規表現の先頭のキャレット（^）は，行の先頭とマッチします。
$	正規表現の末尾のドル記号（$）は，行の最後とマッチします。
.	ピリオド（.）は，任意の1文字とマッチします。
*	アスタリスク（*）が後についた正規表現は，その文字の0回以上の出現とマッチします。たとえば，fo *では，*は o に作用し，f, fo, foo など(f の後に0個以上

のoが続くもの）とマッチしますが，fa などとはマッチしません。

+	プラス（+）が後についた正規表現は，その文字の1回以上の出現とマッチします。fo+は fo，foo などとマッチしますが，f とはマッチしません。
[]	大カッコ[]で囲まれた文字列は，その文字列内の任意の文字とマッチし，他の文字とはマッチしません。カッコ内の最初の文字がキャレット（^）の場合は，その文字列に含まれる文字以外の文字とマッチします。たとえば，[^xyz]は x, y, z とマッチします。また，[xyz]は，a, b などとはマッチしますが，x, y, z とはマッチしません。2つの文字の間にマイナス（-）を置いて，文字の範囲を指定することもできます。これらを，互いに組み合わせることもできます（[a-bd-z?]は，c 以外のすべての小文字と?にマッチすることを意味します）。
¥	円記号（¥）は，次に続く文字が正規表現の演算子ではなく，リテラル文字であることを示すエスケープ文字として機能します。たとえば，¥.は任意の1文字ではなく，ピリオド（.）そのものとマッチするわけです。

注意：上に示した文字のうち，4つ（$. * +）は，大カッコ（[]）の中に置いたときには特別な意味を持ちません。また，^は[の直後に現われない限り特別には扱われません。上であげなかった通常の文字は，同じ文字とマッチします（たとえば>は>と，#は#とマッチします。

ファイル指定

GREP コマンドの3番目の構成要素は**ファイル指定**，すなわちファイルを指定して，探索すべきファイルを GREP に知らせるものです。ファイル指定としては，ファイル名そのもの，あるいは MS-DOS のワイルドカード（?, *）を含むファイル名が使えます。また，ファイル指定の一部として，パス（ドライブ名やディレクトリ名）を入れることもできます。ファイル指定にパスが含まれていない場合は，カレントディレクトリのみが探されます。

なんのファイル指定も指定しない場合，GREP への入力は標準入力からのリダイレクションあるいはパイプによって指定されなければなりません。

使用例と注意

以下に示す例では，GREP のオプションのデフォルトは，すべてのオフになっていることにしています。

例 1

コマンドライン： `grep main( *.c`

マッチする行：
```
main()
mymain(
```

マッチしない行：
```
mymainfunc()
MAIN(i: integer);
```

探索するファイル：　カレントディレクトリにある ＊.C

注意：デフォルトでは，探索の際には大文字小文字は区別されます。

例 2

コマンドライン： `grep -r [^a-z]main¥ *( *.c`

マッチする行：
```
main(i:integer)
main(i,j:integer)
if (main  ()) halt;
```

マッチしない行：
```
mymain()
MAIN(i:integer);
```

探索するファイル：　カレントディレクトリにある ＊.C

注意：GREP が探索する文字列は，前に小文字がついておらず（[^a-z]），後に0個以上の空白(¥ ＊)，さらに左カッコが続いている，main という単語です。空白とタブは，通常コマンドラインの区切りとみなされるので，正規表現の一部に含めるにはハッキリと示す必要があります。この例では，main の後の空白の前にエスケープ文字（¥）をつけています。また，空白を二重引用符（"）で囲んでも（[^a-z]main" " ＊)，同じことになります。

例 3

コマンドライン： `grep -ri [a-c]:¥¥data¥.fil *.c *.inc`

マッチする行：
```
A:¥data.fil
C:¥Data.Fil
B:¥DATA.FIL
```

マッチしない行：
```
d:¥data.fil
a:data.fil
```

探索するファイル：　カレントディレクトリにある＊.C および＊.INC

注意：円記号（¥）とピリオド（.）は通常，特別な意味があるので，これらを探索す

る場合には，直前にエスケープ文字（¥）をつけなければなりません。

例 4

コマンドライン： `grep -ri [^a-z]word[^a-z] *.doc`

マッチする行：

```
every new word must be on a new line.
MY WORD!
word--smallest unit of speech.
In the beginning there was the WORD, and the WORD
```

マッチしない行：

```
Each file has at least 2000 words.
He misspells toward as toward.
```

探索するファイル： カレントディレクトリにある＊.DOC

注意：この指定は，与えられた単語を探索する基本的な方法になっています。

例 5

コマンドライン： `grep -iw word *.doc`

マッチする行：

```
every new word must be on a new line.  However
MY WORD!
word--smallest unit of speech which conveys meaning.
In the beginning there was the WORD, and the WORD
```

マッチしない行：

```
each document contains at least 2000 words.
He seems to continually misspell "toward" as "toward".
```

探索するファイル： カレントドライブ上のルートディレクトリとすべてのサブディレクトリにある＊.DOC

注意：これは，基本的なワードサーチの例です。

例 6

コマンドライン： `grep "search string with spaces" *.doc *.asm a:¥work¥myfile.*`

マッチする行：

```
This is a search string with spaces in it.
```

マッチしない行：

```
THIS IS A SEARCH STRING WITH SPACES IN IT.
This is a search string with many spaces in it.
```

探索するファイル： カレントディレクトリにある＊.DOC と＊.ASM，およびドライブ A の¥WORK というディレクトリにある MYFILE.＊

注意：これは，空白を含む文字列を探索する場合の一例です。

例 7

コマンドライン：

```
grep -rd "[ ,.:?'¥"]"$ ¥*.doc
```

マッチする行：

```
He said hi to me.
Where are you going?
Happening in anticipation of a unique situation,
Examples include the following:
"Many men smoke, but fu man chu."
```

マッチしない行：

```
He said "Hi" to me
Where are you going? I'm headed to the beach this
```

探索するファイル：　カレントドライブのルートディレクトリおよびそのサブディレクトリの中の＊.DOC。

注意：この例では，行の最後にある，．：？’の文字を探索します。ここで大カッコの中の二重引用符は，文字列の終わりを表わす引用符としてではなく，通常の文字として扱われるように，エスケープ文字が前につけられていることに注意してください。また，＄文字が引用文字列の外側に置かれていることにも注意しましょう。このようにすれば，正規表現をいくつかつないで長くすることができるわけです。

例 8

コマンドライン：

```
grep -ild " the " ¥*.doc
(または)  grep -i -l -d " the " ¥*.doc
(または)  grep -il -d " the " ¥*.doc
```

マッチする行：

```
Anyway, this is the time we have
do you think? The main reason we are
```

マッチしない行：

```
He said "Hi" to me just when I
Where are you going? I'll bet you're headed to
```

探索するファイル：　カレントドライブ上のルートディレクトリとすべてのサブディレクトリにある＊.DOC

注意：この例では，大文字小文字を区別せず，マッチする行が少なくとも1行あるファイルの名前を出力するだけです。示した3つのコマンドラインの例は，複数のオプションを指定するいろいろな方法を示しています。

例 9

コマンドライン：　`grep -w[=] = *.c`

マッチする行：

```
i = 5;
j=5;
i += j;
```

マッチしない行：

```
if (i == t) j++;
/* ==================================================*/
```

探索するファイル：　カレントディレクトリの中の＊.C。

注意：この例では，ワードを構成する文字として代入演算子（=）のみを定義しています。この場合，Cの代入演算子はマッチしますが，関係演算子(==)はマッチしないことになります。

BGIOBJ：グラフィックスドライバ/フォント変換ユーティリティ

BGIOBJ は，グラフィックスドライバと文字セット（ストロークフォントファイル）をオブジェクトファイル（.OBJ）に変換するユーティリティです。変換されたドライバおよびフォントは，プログラムにリンクすることができ，実行可能ファイルの一部となります。これは，グラフィックドライバと文字セット（ストロークフォント）を，プログラムの実行時にディスクから読み込む動的ロード機能に加えて提供されています。

ドライバとフォントをプログラムに直接リンクしておくと，実行可能ファイルには必要となるすべての（あるいはほとんどの）ドライバとフォントを含まれるので，実行中にディスク上のドライバやフォントファイルをアクセスする必要がなくなります。ただし，ドライバやフォントをリンクすると，実行可能ファイルのサイズはその分大きくなります。

ドライバやフォントファイルをリンク可能なオブジェクトファイルに変換するには，BGIOBJ.EXE ユーティリティを使用します。

コマンドラインの書式は簡単です。

```
BGIOBJ  <source file>
```

〈*source file*〉には，オブジェクトファイルに変換するドライバまたはフォントファイル名を指定します（拡張子はつけません）。作成されるオブジェクトファイル名は，〈*source file*〉名に拡張子.OBJ をつけたものになります。たとえば，PC98.BGI は PC98.OBJ に，CGA.BGI は CGA.OBJ に，SANS.CHR は SANS.OBJ になります。これらは，デフォルトファイル（BGIOBJ にあらかじめパラメータが登録されているもの）なので，コマンドラインはこのように簡単なものになります。

新しい.OBJ ファイルを GRAPHICS.LIB に追加する

ドライバあるいはフォントのオブジェクトモジュールは，リンカがグラフィックルーチンを取り込む際に，そのオブジェクトモジュールがどこにあるかがわかるように，GRAPHICS.LIB に加えておくとよいでしょう。新しく作成したオブジェクトモジュールを GRAPHICS.LIB に追加しない場合，TC のプロジェクトファイル（.PRJ）の中に，あるいは TCC または TLINK のコマンドラインのファイルリストに，そのオブジェクトモジュール名を加えなければなりません。オブジェクトモジュールを GRAPHICS.LIB に追加するには，次のようにして Turbo ライブラリアン（TLIB）を呼び出します。

```
tlib  graphics + <object filename> [ + <object filename> ...]
```

<*object filename*>には，BGIOBJ.EXEで作成されたオブジェクトファイルの名前（PC98，CGA，GOTHなど）を指定します。拡張子を指定しなければ，.OBJが付け加えられます。時間を節約するために，1つのコマンドラインで複数のファイルを追加することができます。後で示す例を見てください。

ドライバとフォントの登録

ドライバとフォントのオブジェクトモジュールをGRAPHICS.LIBに追加した後は，リンクしたいすべてのドライバとフォントを登録しなければなりません。ドライバとフォントの登録は，プログラムの中で（**initgraph**より前に）**registerbgidriver**や**registerbgifont**を呼び出すことによって行ないます。この呼び出しは，グラフィックスシステムにそのファイルがあることを知らせ，リンカが実行可能ファイルを作成する際にそのファイルを取り込むことを確実にするためのものです。

各登録ルーチン（**register...**）は1つの引数をとります。この引数はGRAPHICS.Hで定義されているシンボル名です。ドライバあるいはフォントの登録が成功した場合，登録ルーチンは負でない値を返します。

次に示す表は，Turbo Cに含まれているドライバとフォントファイルです。また，**registerbgidriver**と**registerbgifont**の引数として用いるシンボル名も示してあります。

ドライバファイル (*.BGI)	registerbgidriver シンボル名	フォントファイル (*.CHR)	registerbgifont シンボル名
■ PC-9801：			
PC98	PC98_driver	TRIP	triplex_font
PC98GRCG	PC98GRCG_driver	LITT	small_font
PC98EGC	PC98EGC_driver	SANS	sansserif_font
		GOTH	gothic_font
■ IBM PC：			
CGA	CGA_driver	TRIP	triplex_font
EGAVGA	EGAVGA_driver	LITT	small_font
HERC	Herc_driver	SANS	sansserif_font
ATT	ATT_driver	GOTH	gothic_font
PC3270	PC3270_driver		
IBM8514	IBM8514_driver		

使用例

ここでは1つのまとまった例を示します。いま，グラフィックスドライバPC98，ゴシックフォント，トリプレックスフォントをオブジェクトモジュールに変換しそれらをプログラムにリンクしたいとします。

1. BGIOBJ.EXEを用いてバイナリファイルをオブジェクトファイルに変換します。これは，次のように独立した3つのコマンドラインで行ないます。

```
bgiobj  pc98
bgiobj  trip
bgiobj  goth
```

これで，3つのファイルPC98.OBJ，TRIP.OBJ，GOTH.OBJが作られます。

2. これらのオブジェクトファイルを，次のTLIBコマンドによってGRAPHICS.LIBに追加します。

```
tlib  graphics +pc98 +trip +goth
```

オブジェクトファイルをGRAPHICS.LIBに追加しない場合は，(Turbo Cの統合環境を使用している場合は)プロジェクトリストに，あるいはTCCのコマンドラインに，オブジェクトファイル名(PC98.OBJ，TRIP.OBJ，GOTH.OBJを加える必要があります。たとえば，TCCの場合は次のようになります。

```
tcc  niftgraf graphics.lib pc98.obj trip.obj goth.obj
```

3. これらのファイルを次のようにグラフィックスのプログラムに登録します。
(リスト17行 C_138)

```
/* ヘッダファイルでは PC98_driver, triplex_font, gothic_font
   の各シンボルが宣言されている. */
#include <graphics.h>

main()
{
    /* 登録およびエラーチェック */
    if (registerbgidriver(PC98_driver) < 0) exit(1);
    if (registerbgifont(triplex_font)  < 0) exit(1);
    if (registerbgifont(gothic_font)   < 0) exit(1);

    /* ... */

    initgraph(....);  /* initgraph は登録後に呼び出さなければならない */

    /* ... */
}
```

ドライバやフォントをリンクしたときに，リンカエラーSegment exceeds 64Kが出た場合は次の節を参照してください。

/F オプション

この節では，いくつかのドライバやフォントファイルをリンクしたときに，リンカエラー Segment exceeds 64K が生じた場合の対処方法を説明します（とくにタイニィ，スモール，コンパクトモデルのプログラムの場合）。

デフォルトでは，BGIOBJ.EXE で作成されたファイルは，すべて同じセグメント（セグメント名は_TEXT）を使用します。このことは，多数のドライバやフォントをリンクする場合や，スモール，コンパクトメモリモデルを使用している場合に問題が生じる原因となります。

この問題を解決するために，1つ以上のドライバやフォントを BGIOBJ の/F オプションで変換することができます。/F オプションをつけると，BGIOBJ はデフォルトのセグメントが，ドライバとフォントによってあふれることがないように，〈*filename*〉_TEXT という形のセグメント名を用います。たとえば，次に示す2つの BGIOBJ コマンドラインは，BGIOBJ に PC98_TEXT と SANS_TEXT というセグメント名を用いることを指示することになります。

```
bgiobj /F pc98.bgi pc98f _PC98_driver_far
bgiobj /F sans
```

/F オプションを用いた場合，BGIOBJ は生成するオブジェクトファイルに F をつけ（PC98F.OBJ，SANSF.OBJ など），**registerfarbgidriver** あるいは **registerfarbgifont** の引数の名前に *_far* をつけます（たとえば *PC98_driver* は *PC98_driver_far* になります）。/F を用いて作成されたファイルに対しては，通常の **registerbgidriver** と **registerbgifont** ではなく，これらの far 用登録ルーチンを使わなければなりません。たとえば次のようにします。

```
if (registerfarbgidriver(PC98_driver_far)  < 0) exit(1);
if (registerfarbgifont(sansserif_font_far) < 0) exit(1);
```

BGIOBJ の高度な機能

この節では，BGIOBJ の高度な機能と，**registerfarbgidriver**，**registerfarbgifont** の説明をします。これらは，熟練したユーザだけが使用すべき機能です。

次に，BGIOBJ.EXE ユーティリティの完全な構文を示します。

```
BGIOBJ  [/F]  <source> <destination> <public name> <seg-name> <seg-class>
```

構成要素	説明
/**F** または -**F**	このオプションは, BGIOBJ.EXE にデフォルトの_TEXT 以外のセグメント名を用いること，パブリック名と生成するファイルの名前を変更することを指示します（これについては前の節を参照してください)。
〈**source**〉	変換するドライバあるいはフォントファイルです。変換するファイルがデフォルトのドライバファイルやフォントファイルでない場合，拡張子（. BGI など）を含めたファイル名を指定する必要があります。
〈**destination**〉	作成されるオブジェクトファイルの名前です。デフォルトのオブジェクトファイル名は〈*source*〉.OBJ で, /F オプションを用いた場合は〈*source*〉F.OBJ になります。
〈**public name**〉	オブジェクトモジュールを取り込むための **registerbgidriver** あるいは **registerbgifont**（またはそれぞれの far 版）の呼び出しで，プログラム内で使用される名前です。 パブリック名はリンカが使用する外部名なので，プログラム内で用いる名前の先頭には下線をつけておくとよいでしょう。プログラムを Pascal の呼び出し慣例で書いている場合は，大文字だけを使用し，下線はつけないでください。
〈**seg-name**〉	オプションのセグメント名です。デフォルトは_TEXT（/F 指定がある場合は〈*filename*〉_TEXT）です。
〈**seg-class**〉	オプションのセグメントクラスです。デフォルトは CODE です。

〈*source*〉以外のパラメータはすべてオプション（省略可能）です。オプションのパラメータを指定する必要がある場合は，指定するパラメータの前にあるすべてのパラメータ名も指定しなければなりません。デフォルトでないユーザ独自のパブリック名を使用する場合は，プログラムの中で，次のどちらかを用いて宣言しなければなりません。

```
void  public_name(void);          /* /F をつけなければ */
                                  /* デフォルトのセグメント名が使われる */
extern int far public_name[];     /* /F をつけた場合 */
                                  /* セグメント名は _TEXT ではない */
```

宣言の中の *public_name* は，BGIOBJ のコマンドラインで指定した〈*public name*〉（パブリック名）です。ヘッダファイル GRAPHICS.H は，デフォルトのドライバとフォントのパブリック名の宣言を含んでいるので，デフォルトのパブリック名を使用する場合は，このような宣言をする必要はありません。

この宣言を行なった後，プログラムで使用しているすべてのドライバとフォントを登録しなければなりません。/F オプションをつけない場合や，デフォルトのセグメント名を変更しない場合は，**registerbgidriver** と **registerbgifont** を使って登録してください。その他の場合は **registerfarbgidriver** と **registerfarbgifont** を使ってください。

次に示すプログラムは，フォントファイルをメモリにロードする例です。

```
/* フォントファイルをメモリにロードする例 */

#include <graphics.h>
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <process.h>
#include <alloc.h>

main()
{
   void     *gothic_fontp;        /* メモリ上のフォント用のバッファを指す */
   int      handle;               /* I/O に使用するファイルハンドル        */
   unsigned fsize;                /* ファイル（およびバッファ）のサイズ    */

   int errorcode;
   int graphdriver;
   int graphmode;

   /* フォントファイルをオープンする */
   handle = open("GOTH.CHR", O_RDONLY|O_BINARY);
   if (handle == -1)
   {
      printf("unable to open font file 'GOTH.CHR'¥n");
      exit(1);
   }

   /* ファイルのサイズを求める */
   fsize = filelength(handle);


/* 次ページに続く */
```

```
    /* バッファを割り当てる */
    gothic_fontp = malloc(fsize);
    if  (gothic_fontp == NULL)
    {
        printf("unable to allocate memory for font file 'GOTH.CHR'¥n");
       exit(1);
    }

    /* フォントをメモリに読み込む */
    if (read(handle, gothic_fontp, fsize) !=fsize)
    {
       printf("unable to read font file 'GOTH.CHR'¥n");
       exit(1);
    }

    /* フォントファイルをクローズする */
    close (handle);

    /* フォントを登録する */
    if (registerfarbgifont(gothic_fontp) !=GOTHIC_FONT)
    {
       printf("unable to register font file 'GOTH.CHR'¥n);
       exit(1);
    }

    /* 検出を行なって，グラフィックスを初期化する */
    graphdriver = DETECT;
    initgraph(&graphdriver, &graphmode, "..");
    errorcode = graphresult();
    if (errorcode != grOk)
    {
      printf("graphics error: %s¥n", grapherrormsg(errorcode));
      exit(1);
    }
    settingjustify(CENTER_TEXT, CENTER_TEXT);
    settextstyle(GOTHIC_FONT, HORIZ_DIR, 4);
    outtextxy( getmaxx() / 2, getmaxy() / 2,
               "Borland Graphics Interface (BGI)");

    /* どれかキーを押すと終了する */
    getch();

    /* グラフィックスシステムを終了する */
    closegraph();
    return (0);
}
```

OBJXREF：オブジェクトモジュール相互参照ユーティリティ

OBJXREFは，リスト中のオブジェクトファイルやライブラリファイルの内容を調べて，その結果をレポートするユーティリティです。レポートには2つの形式があります。1つの形式は，**パブリック名**の定義と，それらの参照関係を示したもので，もう1つの形式は，**オブジェクトモジュール**によって定義されるセグメントの大きさを示したものです。

パブリック名は，**グローバル変数**と**関数名**の2つのカテゴリに分類することができます。「**OBJXREFのレポート例**」の節で示しているTEST1.CとTEST2.Cというファイルは，パブリック名の定義と，それらに対する外部参照関係を例示しています。

オブジェクトモジュールは，TC，TCC，TASMによって作られるオブジェクトファイル（.OBJ）です。ライブラリファイル（.LIB）というのは，多くのオブジェクトモジュールを集めたものです。TCで作られるオブジェクトモジュールの名前は，コンパイルされるソースファイル（.C）と同じ名前になります。TCCでコンパイルする場合も同じですが，-oというTCCコマンドラインオプションを用いると，出力ファイル名を違う名前にすることができます。

OBJXREFのコマンドライン

OBJXREFのコマンドラインは，コマンド名OBJXREFと，それに続く一連のオプション，オブジェクトファイル名やライブラリファイル名の並びから構成されます。区切り記号は，空白またはタブで，構文は次のようになります。

```
OBJXREF [オプション] ファイル名 [ファイル名 ...]
```

コマンドラインオプションによって，OBJXREFが出力するレポートの種類と，内容の詳しさの程度を指定します。詳しくは，後で述べる「**OBJXREFコマンドラインオプション**」の節を参照してください。

オプションは，スラッシュ（/）で始まる1文字または2文字のオプション名からなります。

オブジェクトファイル名とライブラリファイル名は，コマンドラインあるいはレスポンスファイルの中で指定することができます。コマンドラインで指定する場合は，ファイル名は空白かタブで区切ります。.OBJファイルとして指定されたオブジェクトファイルは，すべてレポートに含められます。ただし，TLINKと同様に，OBJXREFは.OBJファイル

によって，あるいは前に.LIB ファイルから取り込まれたモジュールによって参照されるパブリック名を含むモジュールしか取り込みません。

一般に，スタートアップ.OBJ ファイルとCライブラリを含めて，プログラムを正しくリンクするのに必要なすべての.OBJ ファイルと.LIB ファイルを指定しておくべきです。

ファイル名には，ドライブやディレクトリパスを含めてもかまいません。DOSのワイルドカード文字（?と*）を使って，複数のファイルを指定することもできます。ファイル名としては，オブジェクトファイル（.OBJ），あるいはライブラリファイル（.LIB）を指定します（拡張子がないときは.OBJ が仮定されます）。

オプションとファイル名は，コマンドラインの中ではどのような順序で置いてもかまいません。

OBJXREF のレポートは，DOSの標準出力に書き出されます。デフォルトは画面となっています。DOSのリダイレクション記号（>）を使って，プリンタに出力（>PRN のように書きます）したり，ファイルに出力（>lstfile のように書きます）したりすることができます。

ファイル名やオプションをまったく指定せずにOBJXREF を実行すると，使用できるすべてのオプションの要約が示されます。

OBJXREF コマンドラインオプション

OBJXREF のコマンドラインオプションは，**コントロールオプション**と**レポートオプション**の2つに分類することができます。

コントロールオプション

コントロールオプションによって，OBJXREF のデフォルトの動作を変えることができます（デフォルトでは，以下のオプションがOn になっていません）。

/I　　パブリック名における大文字小文字の違いを区別しません。TLINK を/C オプション（大文字小文字の違いを区別するもの）なしで実行する場合に，このオプションを使ってください。

/F　　ライブラリ全体を取り込みます。たとえOBJXREF によって処理されるオブジェクトモジュールによって，その中に含まれるパブリック名が参照されていなくても，指定された.LIB ファイルの中のすべてのモジュールが取り込まれます。このオプションによって，ライブラリファイルの内容全体についての情報を知ることができます（「**OBJXREF コマンドの例**」の節の例4を参照してください）。

/V　　詳細な出力を行ないます。読み込まれたファイル名全部を表示し，また，パブリック名，モジュール，セグメント，クラスの総計も示します。

/Z　　長さが0のセグメント定義も取り込みます。オブジェクトモジュールによっては，その中にまったく領域を割り当てていないセグメントを定義しているものもあります。長さが0のセグメント定義まで表示するとモジュールサイズのレポートは見にくくなりますが，セグメントのすべての定義を取り除こうとする場合には，このオプションは役に立ちます。

レポートオプション

レポートオプションは，出力されるレポートの種類，およびOBJXREFが出力する情報の詳しさを決めます。

/RC（クラスタイプ別のレポート）

セグメントのクラスタイプごとにわけられて，モジュールが大きさの順で並べられます。

/RM（モジュール別のレポート）

定義モジュールごとにわけられて，パブリック名が名前の順で並べられます。

/RP（パブリック名順のレポート）

パブリック名が順番に並べられ，その後ろにはそれが定義されているモジュール名がつきます。

/RR（参照によるレポート）

パブリック名が名前の順で並べられ，それを参照するモジュールも名前の順で並べられます（これは，レポートオプションが設定されていないときのデフォルトです）。

/RS（モジュールサイズのレポート）

セグメント名ごとにわけられて，モジュールの大きさの順で並べられます。

/RU（参照されていないシンボルのレポート）

参照されていないパブリック名が，定義モジュールごとにわけられて，名前の順で並べられます。

/RV（詳細なレポート）

OBJXREF は，すべての形式のレポートを出力します。

/RX（外部参照のレポート）

モジュール名ごとにわけられて，その中に含まれている外部参照名を名前の順で並べられます。

.C ファイルの中で定義されたパブリック名は，コンパイル時に-U-オプションをつけないでコンパイルされていれば，レポート中では先頭に下線がつけられて示されます（**main** は，_main のように表示されるのです）。

レスポンスファイル

コマンドラインの長さは，DOS によって最大127文字までに制限されています。オプションやファイル名のリストがこの制限を越える場合，これらのファイル名を**レスポンスファイル**の中に書かなければなりません。

レスポンスファイルは，エディタで編集することのできるテキストファイルです。他の Turbo C プログラム（たとえば TLINK）に対する，プログラム作成用ファイルリストをすでに用意しているかもしれないので，OBJXREF はいくつかの種類のレスポンスファイル形式を認識することができます。

レスポンスファイルは，コマンドラインから以下に示すオプションのうちの1つを用いて呼び出すことができます。そのオプションとレスポンスファイル名の間にスペースを入れてはいけません（/L resp ではなく/Lresp です）。

自由形式のレスポンスファイル

レスポンスファイルは，テキストエディタを使って自由形式で作ることができます。ファイルの内容は，.EXE ファイルの作成に必要なすべての.OBJ ファイルと.LIB ファイル名をそのまま書き並べたものです。

OBJXREF で，この自由形式のレスポンスファイルを用いるには，ファイル名の先頭に単価記号（@）をつけて指定します。コマンドラインの他の要素とは，空白またはタブで区切ります。

```
@filename @filename ...
```

注意：レスポンスファイル中で拡張子を省略して指定されたファイル名は.OBJ ファイルとみなされます。

プロジェクトファイル

TC によって生成された形式のプロジェクトファイルも，レスポンスファイルとして用いることができます。コマンドラインに書くときは，プロジェクトファイル名の前に次のように/P を入れます。

```
/Pfilename
```

ファイル名に拡張子が含まれないときは，.PRJ という拡張子が仮定されます。

プロジェクトファイル中で拡張子.C がついたファイルや拡張子がないファイル名は，対応する.OBJ ファイル名を指定していると解釈されます。カッコの中に指定した依存ファイルは，OBJXREF によって無視されるので，取り除く必要はありません。

注意：.PRJ ファイル中にあるファイルリストだけでは，完全なプログラムを指定していることにはなりません。スタートアップファイル（C0*x*.OBJ）と Turbo C ライブラリファイル (math*x*.lib, emu.lib, C*x*.lib など）も指定しなければなりません。さらに，OBJXREF が.OBJ ファイルを探索するディレクトリを指定するために，/O オプションを使う必要がある場合があるはずです。

リンカのレスポンスファイル

TLINK のレスポンスファイルのフォーマットで書かれたファイルも，OBJXREF のレスポンスファイルとして使うことができます。コマンドラインでリンカ用のレスポンスファイルを呼ぶときは，次のように先頭に/L と書きます。

```
/Lfilename
```

これらのファイル形式をどのように使うかについては，「**OBJXREF の使い方の例**」の例 **2** を参照してください。

/O コマンド

OBJXREF にカレントディレクトリ以外のディレクトリにある.OBJ ファイルを探させるためには，コマンドラインの中で次のように先頭に/O を付けてディレクトリの名前を書きます。

```
/Omyobjdir
```

/N コマンド

OBJXREF にレポートさせたいモジュール，セグメント，クラス，パブリック名を，コマンドラインにおいて希望の名前の先頭に/N をつけて指定することができます。たとえば次のコマンドラインは，

```
OBJXREF <filelist> /RM /NC0
```

C0という名前のモジュールについての情報だけをレポートするよう OBJXREF に指定しています。

OBJXREF のレポート例

いま Turbo C のディレクトリに，2つのソースファイルがあるとします。そして，それらのソースファイルからコンパイルしたオブジェクトファイルについての OBJXREF レポートを作成したいということにします。2つのソースファイルの名前は TEST1.C と TEST2.C で，次のようなファイルです。

TEST1.C：

```
int i1;                         /* i1 を定義 */
extern int i2;                  /* i2 を参照 */
static int i3;                  /* パブリック名ではない */
extern void look(void);         /* look を参照 */

void main(void)                 /* main の定義 */
{
    int i4;                     /* パブリック名ではない */

    look();                     /* look を参照 */
}
```

TEST2.C：

```
#include <process.h>
extern int i1;                    /* i1 を参照 */
int i2;                           /* i2 を定義 */

void look(void)                   /* look を定義 */
{
    exit(i1);                     /* exit を参照し，i1を渡す */
}
```

上のファイルをコンパイルしたオブジェクトモジュールを TEST1.OBJ および TEST2.OBJ とします。OBJXREF に，これらの.OBJ ファイルに対してどのようなレポートを要求するかは，コマンドラインにおいてファイル名を指定するとともに，/R の後にレポートの種類を表わす文字をつけて指定します。

注意：以下に示す例では，出力の一部分だけしか示されていません。

パブリック名順のレポート（/RP）

パブリック名順のレポートは，レポートの対象になっているオブジェクトモジュールの中で定義されているパブリック名の順で示します。パブリック名の後ろにはそれが定義されているモジュール名がつきます。

次のようなコマンドラインを入力すると，

```
OBJXREF /RP test1 test2
```

OBJXREF は，以下のようなレポートを出力します。

```
SYNBOL      DEFIND IN
_i1         TEST1
_i2         TEST2
_look       TEST2
_main       TEST1
```

モジュール別のレポート（/RM）

モジュール別のレポートは，レポートの対象になっているオブジェクトモジュールごとにわけられ，そこで定義されているパブリック名が名前の順で並べられたものです。

次のようなコマンドラインを入力すると，

```
OBJXREF /RM test1 test2
```

OBJXREFは，以下のようなレポートを出力します。

```
MODULE: TEST1 defines the following symbols:
     public: _i1
     public: _main
MODULE: TEST2 defines the following symbols:
     public: _i2
     public: _look
```

参照によるレポート（/RR）（デフォルト）

参照によるレポートは，パブリック名と，それが定義されているモジュール名（カッコ内に表示）を1行で表わしたものを順に並べたものです。そのパブリック名を参照しているモジュール名は，インデント（段下げ）されて表示されます。

次のようなコマンドラインを入力すると，

```
OBJXREF /RR C0 test1 test2 CS.LIB
```

OBJXREFは，以下のようなレポートを出力します。

```
_exit (EXIT)
     C0
     TEST2
_i1 (TEST1)
     TEST2
_i2 (TEST2)
_look (TEST2)
     TEST1
_main (TEST1)
     C0
```

外部参照のレポート（/RX）

外部参照のレポートは，モジュールごとに，各モジュールの中に含まれる外部参照のリストを出力します。

次のようなコマンドラインを入力すると，

```
OBJXREF /RX test1 test2 CS.LIB
```

OBJXREF は，以下のようなレポートを出力します。

```
MODULE: C0 references the following symbols:
     _main
MODULE: TEST1 references the following symbols:
     _i2
     _look
MODULE: C0 references the following symbols:
     _exit
     _i1
```

モジュールサイズのレポート（/RS）

モジュールサイズのレポートは，セグメント名の後に，そのセグメントを定義しているモジュールのリストを並べたものです。バイト単位のサイズは10進と16進で与えられます。"uninitialized"という語は，セグメントの中で定義されているどのシンボルにも初期値が代入されていないときに現われます。.ASM ファイルの中で絶対アドレスで定義されているセグメントは，セグメントサイズの左側に Abs というフラグがつけられます。

次のようなコマンドラインを入力すると，

```
OBJXREF /RS test1 test2
```

OBJXREF は，以下のようなレポートを出力します。

```
TEST1_TEXT
        6 (00006h)    TEST1
        6 (00006h)    total
TEST2_TEXT
       10 (0000Ah)    TEST2
       10 (0000Ah)    total
_BSS
        4 (00004h)    TEST1, uninitialized
        2 (00002h)    TEST2, uninitialized
        6 (00006h)    total
```

クラスタイプ別のレポート（/RC）

クラスタイプ別のレポートでは，セグメントサイズ定義がセグメントクラスごとにリストされます。CODEというクラスには命令が含まれ，DATAというクラスには初期化したデータが含まれ，BSSというクラスには初期化されていないデータが含まれます。クラスタイプを持たないセグメントは，No class typeという分類に入ります。
次のようなコマンドラインを入力すると，

```
OBLXREF /RC C0 test1 test2 CS.LIB
```

OBJXREFは，以下のようなレポートを出力します。

```
BSS
         4  (00004h)       TEST1
         2  (00002h)       TEST2
       ...
       132  (00084h)       total
CODE
         6  (00006h)       TEST1
        10  (0000Ah)       TEST2
        16  (00010h)       total
DATA
       143  (0008Fh)       C0
       143  (0008Fh)       total
```

参照されないシンボル名のレポート（/RU）

参照されないシンボル名のレポートは，他のモジュールの中で参照されていないパブリック名を定義しているモジュールをリストします。そのようなシンボルは次のうちいずれかです。

■定義しているモジュール内でのみ参照されていて，パブリック宣言する必要のないもの（このような場合，そのモジュールがCファイルであるときは，キーワードstatic

を宣言に付加するべきです。また，モジュールが TASM ファイルであるときは，単にパブリック宣言を取り除けばよいのです)。

■使われないもの（したがって，それを削除して，コードやデータ領域を節約するべきです)。

次のようなコマンドラインを入力すると，

```
OBJXREF /RU test1 test2
```

OBJXREF は，以下のようなレポートを出力します。

```
MODULE: TEST2 defines the unreferenced symbol _i2
```

詳細なレポート（/RV）

コマンドラインのオプションを，/RV とすると，すべての形式のレポートが出力されます。

OBJXREF の使用例

以下の例では，対象となるファイルはデフォルトドライブのカレントディレクトリにあり，Turbo C スタートアップファイル（C0*x*.OBJ）とライブラリファイルは¥TURBOC¥LIB ディレクトリにあるとします。

例 1：　`A>OBJXREF ¥turboc¥lib¥c0l test1 test2 ¥turboc¥lib¥cl.lib`

TEST1.OBJ，TEST2.OBJ ファイルに加えて，Turbo C のスタートアップファイル¥TURBOC¥LIB¥C0L.OBJ と，ライブラリファイル¥TURBOC¥LIB¥CL.LIB が指定されています。レポート形式は指定されていないので，デフォルトの参照によるレポート（パブリック名とそれを参照しているモジュール）が出力されます。

例 2：　`A>OBJXRF /RV /Ltest1.arf`

TLINK のレスポンスファイル TEST1.ARF は，例1のコマンドラインとほぼ同じファイルを含んでいます。/RV オプションが指定されているので，すべての形式のレポートが出力されます。TEST1.ARF ファイルは，以下のような内容になっています。

```
¥turboc¥lib¥c01
test1 test2
test1.exe
test1.map
¥turboc¥lib¥cl
```

例 3：　`C>OBJXREF /RC B:c0s /Ptest1 @libs`

TC のプロジェクトファイル TEST1.PRJ によって，TEST1.OBJ と TEST2.OBJ が指定されます。レスポンスファイル@libs によって，ドライブ B 中ののディスク上にあるライブラリが指定されます。TEST1.PRJ ファイルは，次のような内容になっています。

```
test1
test2.c
```

また，LIBS ファイルは，次のような内容になっています。

```
b:maths.lib b:emu.lib b:cs.lib
```

指定すべきスタートアップファイルやライブラリファイルは，コンパイル時に使われるメモリモデルや浮動小数点オプションに依存します。/RC オプションによってクラスタイプごとにわけたレポートが出力されます。

例 4：　`A>OBJXREF /F /RV ¥turboc¥lib¥cs.lib`

この例は，Turbo C のライブラリファイル CS.LIB の中のすべてのモジュールについてのレポートを出力します。指定したファイルが完全なプログラムとならない場合でも，OBJXREF は有用なレポートを出力します。/F コマンドは，CS.LIB に含まれているすべてのモジュールをレポートの対象とすることを意味しています。

OBJXREFのエラーメッセージと警告

OBJXREFは，エラーメッセージと警告という2種類の診断メッセージを出力します。

エラーメッセージ

Out of memory

OBJXREFは，相互参照レポート作成の作業をRAMメモリ上で行ないます。同じファイルリストをTLINKが正常にリンクできた場合でも，OBJXREFではメモリ不足となる場合があります。メモリが不足したた場合，OBJXREFは異常終了します。メモリ常駐のプログラムを取り除いて使えるメモリ空間を増やすか，RAMメモリを増設してください。

警告

WARNING： Unable to open input file rrrr

*rrrr*という名前のファイルを見つけることができなかったか，あるいはオープンすることができません。OBJXREFは次のファイルの処理に進みます。

WARNING： Unknown option - oooo

OBJXREFが*oooo*で指定されたオプション名を認識できません。OBJXREFは，そのオプションを無視します。

WARNING： Unresolved symbol nnnn in module mmmm

モジュール*mmmm*で参照されているパブリック名*nnnn*は，指定された.OBJファイルあるいは.LIBファイルの中では定義されていません。OBJXREFは，出力するすべてのレポートにおいて，そのシンボルは参照されているが定義されていないという意味のフラグを表示します。

WARNING： Invalid file specification ffff

ファイル*ffff*の中に正しくない部分があります。OBJXREFは次のファイルを処理します。

WARNING： No files matching ffff

コマンドライン上か，レスポンスファイル中の *ffff* という名前のファイルが見つからないか，オープンできません。OBJXREF は，そのファイルを飛ばして次のファイルを処理します。

WARNING： Symbol nnnn defined in mmmm1 duplicated in mmmm2

パブリック名 *nnnn* がモジュール *mmmm1* と *mmmm2* の両方で定義されています。OBJXREF は2番目の定義を無視します。

付録 D

TCINST：Turbo C のカスタマイズ

TCINST は，Turbo C のインストレーションプログラムです。これを使って，Turbo C の統合開発環境 TC.EXE をカスタマイズします。TCINST によって，画面サイズ，エディタの編集モード，メニューの表示色，デフォルトディレクトリといった TC 使用環境のさまざまなデフォルト値を設定することができます。TCINST は，TC.EXE ファイル内のデフォルト値を直接書き換えます。

TCINST では，以下のようなことが行なえます。

■デフォルトのプライマリファイルおよびプロジェクト名の指定
■インクルードファイル，ライブラリ，コンフィギュレーション，ヘルプファイル，ピックファイルのあるディレクトリ，出力を行なうディレクトリパスの設定
■統合デバッガのデフォルト設定の選択
■エディタのコマンドキーの変更
■エディタのデフォルトと画面表示形式の設定
■デフォルトの画面表示モードの設定
■画面表示色の変更
■ Edit ウィンドウ，Message/Watch ウィンドウのサイズの変更

なお，Turbo C はそのままで実行可能なので，上のようなことが必要なければ TCINST を実行するる必要はありません。第1章にあるとおり，マスターディスクから作業用のフロッピィあるいはハードディスクにインストールすれば，すぐに実行することができます。TC.EXE にあらかじめセットされているデフォルトを直接変更したい場合には，TCINST によって手軽に変更することができます。

注意：TCINST のメニューは，TC 統合開発環境のメニューによく似ています。同じメニュー項目の機能の詳細については，第5章の該当する説明を参照してください。

注意：TCINSTによってインストールしたオプションで，TC.EXEのメニューにも一致する項目があるオプションは，統合環境のメニューからその項目を変更したり，異なる設定内容が書き込まれたコンフィギュレーションファイルをロードすれば，その内容に変更されます。

TCINSTの起動

TCINSTの構文は次のとおりです。

```
tcinst [オプション] [パス名]
```

オプションとパス名は，どちらも省略可能です。パス名を指定しなかった場合には，TCINSTはカレントディレクトリからTC.EXEを探します。指定した場合はそのパス名を使います。

オプションには以下のものがあります。

■ PC-9801：

/8　表示色を設定するためのカラーパレットを強制的に8色モードにします。TCINSTは起動時にマシンの種類を調べて，Set colorsオプションで使用するカラーパレットを決めます。4096中16色を使える場合には，標準の8色ではなく16色のパレットになります。ただし，本体が16色を表示できるもの（VXなど）であってもアナログCRTを接続していない場合には，表示できない色があります。このような場合のために/8オプションが用意されています。

■ IBM PC：

/c　TCINSTの中の表示がカラーになります（これがデフォルトです）。

/b　TCINSTの中の表示を白黒で行ないます。

注意：TCINSTは，システム上にある複数のTurbo Cのコピーに対してカスタマイズを行なうことができます。TC.EXEの各コピーには，それぞれ別のファイル名をつけることができます。この場合に必要なのは，TCINSTを起動するときに，カスタマイズするTC.EXEのコピーのパス名（ディレクトリ名およびファイル名）を指定するだけです。次のようにするわけです。

```
tcinst tc.exe
tcinst /b ..¥..¥bwtc.exe
tcinst /c c:¥borland¥colortc.exe
```

このようにして，Turbo Cの複数のコピーに対して，エディタのコマンドキーの違うもの，メニューの表示色の違うもの，というようにさまざまなバージョンを作ることができます。

TCINST メインメニュー (Installation Menu)

はじめに，次のような TCINST のメインメニュー（Installation Menu）が画面に現われます。

図 D.1　TCINST の Installation Menu

```
┌ Installation Menu ┐
│ Compile           │
│ Project           │
│ Options           │
│ Debug             │
│ Editor commands   │
│ Mode for display  │
│ Set colors        │
│ Resize windows    │
│ Quit/save         │
└───────────────────┘

Turbo C Installation Program 2.0
```

- ■ Compile オプションでは，コンパイルされるプライマリファイルのデフォルト名を指定することができます。
- ■ Project オプションでは，プロジェクトファイルのデフォルト名を指定し，プロジェクトをコンパイルする際のさまざまなデフォルト値をセットすることができます。
- ■ Options コマンドを使うと；数多くの機能，たとえばメモリモデル，最適化レベル，エラーメッセージ，リンカと環境設定，ヘッダファイルやライブラリファイルが入っているディレクトリパスのデフォルト値を変更することができます。
- ■ Debug オプションは，統合デバッガに対する Source debugging や Display swapping のデフォルトを設定するのに使います。
- ■ Editor commands オプションでは，TC エディタのキーストロークコマンドを自分の好みに合わせて再割り当てすること（カスタマイズ）ができます。

■ Mode for display によって，Turbo C の画面表示モードを指定することができます。また，IBM PC では使用するビデオアダプタが“雪降り (snowy)”であるかどうかを指定することができます。

■ Set colors メニューを使うと，TC 統合環境のほとんどすべての部分の色を自分の好みに合わせてカスタマイズすることができます。

■ Resize windows オプションによって，Edit ウィンドウや Message/Watch ウィンドウの大きさを変えることができます。

■ Quit/save オプションを選択すると，統合開発環境に対して行なった変更が記憶され，制御は MS-DOS レベルに戻ります。

メニュー中の各項目を選択するには，オプションのハイライト表示されている最初の文字をタイプします。たとえば Set colors オプションを選択するには，***S*** をタイプします。あるいは，上下矢印キーを使ってハイライトバーを選択したい項目まで動かして，**リターン**キーを押します。

ESC キーを押すと（必要なら複数回），サブメニューからメインメニューに戻ります。

Compile メニュー

Compile メニューには，**Primary file** というオプションが1つだけあります。そのオプションを選択すると，プロンプトボックスが現われ，コンパイル・リンクの対象となるソースファイル（プライマリファイル）のデフォルト名を入力することができます。このオプションは，複数のヘッダファイルを取り込んでいる単一ファイルプログラムの場合に，ヘッダファイルの修正を行ないながら何度もコンパイルを行なうようなときにたいへん便利です。

Project メニュー

Project メニューは，プロジェクトファイルのデフォルト名を指定したり，プロジェクトをコンパイルやリンクする場合の諸機能に対するデフォルトを設定したりするために使用します。

Project name

このオプションを選択するとプロンプトボックスが現われ，その中にプロジェクトファイルのデフォルト名を指定することができます。拡張子の.PRJ は，省略することができます。

Break make on メニュー

このメニューでは，プログラムをコンパイルする際に Make を止めるデフォルトの条件を設定します。**Warning**, **Errors**, **Fatal** errors, **Link** の中から選びます。

Auto dependencies

このオプションは，Auto dependencies トグルスイッチを On にするか，Off にするかのデフォルトを決めます。

Clear project

このオプションは，以前に指定したプロジェクトファイルの名前を取り消すのに使います。したがって，このオプションを使うことで，プロジェクトファイルの名前を変更することができます。

Options メニュー

Options メニューによって，いろいろな機能のデフォルトを設定し，統合環境がどのように働くかを決めることができます。

Compiler メニュー

Options メニュー中のこのオプションは，ハードウェア構成を指定したり，メモリモデル，コードの最適化レベル，診断メッセージのコントロール，マクロ定義などに関するデフォルトを設定するのに使います。

Model

Turbo C のデフォルトのメモリモデル（メモリのアドレシング方法）を変更するのに使います。選択できるオプションは，**Tiny**, **Small**, **Compact**, **Medium**, **Large**, **Huge** のいずれかです。メモリモデルの詳細については第12章を参照してください。

Defines

このオプションを選択するとプロンプトボックスが現われ，その中にマクロ定義を書き込むことができます。これは，TC においてデフォルトで使えるマクロとなります。

Code generation メニュー

このメニューの中にある選択要素によって，コンパイラがどのようにソースコードをコンパイルするかのデフォルトを設定することができます。

Calling convention：C あるいは Pascal の関数呼び出しシーケンス。

Instruction set：8088/8086，または80186/80286。

Floating point：8087/80287，Emulation，または None。

Default char type：Signed または Unsigned。

Alignment：Word アラインメントまたは Byte アラインメント。

Generate underbars：On または Off。

Merge duplicate strings：On または Off。

Standard stack frame：On または Off（注意：デバッガのもとでプログラムを走らせるときは，このオプションは On にすべきです）。

Test stack overflow：On または Off。

Line numbers：On または Off。

OBJ debug information：On または Off。

Optimization メニュー

このメニューのオプションは，ソースコードをコンパイルするときの最適化に関するデフォルトを設定するのに使います。

Optimize for：Size または Speed

Use register variables：On または Off

Register optimization：On または Off

Jump optimization：On または Off（**注意**：デバッガのもとでプログラムを走らせるときは，このオプションは Off にすべきです）。

Source メニュー

このメニューでは，識別名の長さ，ネストしたコメントを許すかどうか，拡張のキーワードを認識するか，それとも ANSI キーワードしか認識しないか，などのデフォルトを指定します。

Errors メニュー

このメニューは，次のような目的に使われます。

■コンパイルをいつ中止するかを決めるエラーや警告の数のデフォルト値（0から255）を設定する。
■警告メッセージを表示するかどうかを決める。
■表示するエラー，警告メッセージを選択する（選択の仕方は，それぞれが On/Off のトグルスイッチになっています）。
4つのタイプに分類され，それぞれがメニュー形式で選択できます。

· Portability warnings
· ANSI violations
· Common errors
· Less common errors

Names メニュー

このメニューによって，**Code**，**Data**，**BBS** の各セクションに対して，セグメント，グループ，クラスのデフォルト名を設定することができます。メニュー中の項目を選択すると，次のメニューを選択するとアスタリスク（＊）が現われますが，これはコンパイラにデフォルト名を使用するよう指示するものです。

Turbo C に習熟し，第12章をよく理解した方以外は，このオプションを変更してはいけません。

Linker メニュー

Linker メニューは，プログラムがライブラリルーチンとどのようにリンクされるかに関係するデフォルトを設定するために使います。これらの設定に関する詳細について，付録 B を参照してください。

Map file

このオプションは，マップファイルに対するデフォルト形式決めるものです。**Off**, **Segments**, **Publics**, **Detailed** のいずれかを選択することができます。

Initialize segments

On または Off の選択ができます。このトグルスイッチを On にすると，リンカは初期化されていないセグメントを初期化します。

Default libraries

On または Off の選択ができます。このオプションは，デフォルトライブラリのリストをオブジェクトファイルの中に書き込むような他のコンパイラで作ったモジュールをリンク

する際に使用します。このオプションがOnのときは，リンカはTurbo C自身のライブラリと同様に，このリスト中のライブラリからも未定義ルーチンを探そうとします。Offのときは，Turbo C自身のライブラリの中でのみ未定義ルーチンを探し，.OBJファイル中のデフォルトライブラリは無視します。

Graphics library

リンカがBGIグラフィックライブラリ関数をリンクするかどうかをコントロールします。デフォルトはOnで，OffにするとリンカはGRAPHICS.LIBを探さなくなります。

Warn duplicate symbols

オブジェクトファイルとライブラリファイルの中でシンボルが重複して使われている場合に，リンカが警告メッセージを出すかどうかを，OnあるいはOffで指定します。

Stack warning

no stackという警告メッセージをリンカが出力するかどうかをOn/Offで指定します。

Case-sensitive link

リンク時に大文字と小文字を区別するかどうかをOn/Offで指定します。C言語は大文字と小文字を区別する言語なので，通常このオプションはOnになっています。

Environment メニュー

このEnvironmentメニューによって，Turbo Cの作業環境を自分の好きなように変えることができます。

各オプションの選択方法は画面の一番下に表示されます。Turbo Cの使用環境を自分の好みに応じて変更し，その環境をTurbo Cにセーブすることができます。もちろん，これらの設定は，統合環境のOptions/Environmentメニューから，あるいはTurbo Cエディタの中からでも変更することができます。

Message tracking

このオプションは，コンパイル時の構文エラーを追跡する範囲を決めます。Current file, All files, Off のいずれかを選択します。

Keep Messages

On か Off を選択します。このオプションによって，前回のコンパイル時のエラーメッセージを Message ウィンドウに保存しておくかどうかを決めます。

Config auto save

On か Off を選択します。このオプションを On にすると，プログラムの実行や DOS のシェルに移行するとき，統合環境を終了するときに，Turbo C は自動的に現在の設定内容を（まだセーブしていなければ）コンフィギュレーションファイルにセーブします。

Edit auto save

On か Off を選択します。このオプションを On にすると，プログラムを修正した場合，そのプログラムを実行するときや，DOS のシェルに移行するときに，Turbo C は自動的に現在のプログラムをセーブします。

Backup source files

On か Off を選択します。このオプションを On にすると，プログラムをセーブするときに，Turbo C は自動的にバックアップファイルを作ります。

Zoomed windows

On か Off を選択します。このオプションを On にすると，TC 開始時のアクティブウィンドウ (Edit または Message/Watch) が画面全体に拡大されます。このオプションが Off にすると，デフォルトでは2つのウィンドウが表示されます。

Palette set up(PC-9801のみ)

TCの起動時に，TCが保持しているパレット情報で，ハードウェアパレットを初期化するかどうかを指定します。Set colorsメニューのPalette customizeで設定されたパレット情報はTC.EXEに書き込まれまるとともに，その時点でハードウェアパレットの内容を変更しますが，マシンをリセットしたり一度電源を切って再び起動したときには，パレットはマシンのデフォルトに戻ってしまいます。TCのパレット情報を必らず使用するようにさせるためには，この項目をOnにセットしておく必要があります。

Full graphics save(IBM PCのみ)

グラフィックス画面をセーブするために，TCは8Kバイトのメモリをパレット用のバッファとして確保しています。テキスト画面しか使わない場合は，Full graphics saveをOffにして，このメモリ空間をTurbo Cに開放することができます。バッファ領域はTCの起動時に確保されるため，このオプションは，TCINSTコマンドにおいてのみ指定可能で，統合開発環境の中では指定できません。

Screen sizeメニュー

Screen sizeメニューによって，統合環境の画面の行数を設定することができます。

■ PC-9801

標準の25行，拡張23行，あるいは圧縮46行に設定することができます。
Mode for displayメニューで，Text onlyを選択している場合には，拡張23行と圧縮46行は選択できません。

Standard 25 line：

テキスト画面による25行の表示です。この場合，Set colorsオプションで指定できる画面の表示色は，8色およびそのリバースの中から指定することになります。

Extended 23 line：

グラフィック画面による23行の表示です。この場合，Set colorsオプションで指定できる画面の表示色は，8色あるいは16色の中からフォア/バックグラウンドそれぞれを指定することができます。

8色/16色のどちらになるかは，使用するマシンによって決まります。PC-9801UVやそれ以降の4096色中16色を表示できるマシン，および16色ボードを装着したマシンにアナログ RGB ディスプレイを接続している場合には，16色を使用することができます。最初期型の PC-9801や PC-9801E/F などで16色ボードを装着していない場合や，16色表示可能なマシンでもデジタル RGB ディスプレイを接続している場合には，8色しか使用できません。Set colors オプションの説明も参照してください。

High density 46 line：

グラフィック画面による46行の表示です。Set colors オプションで指定できる画面の表示色は，Extended 23 line の場合と同じです。

■ **IBM PC**

25行，あるいは43/50行に設定することができます。

25 lines display：

これは，標準的な PC のディスプレイ（25行×80桁）の場合です。MDA（モノクロディスプレイアダプタ），あるいは CGA（カラーグラフィックスアダプタ）が装備されているシステムでは，このスクリーンサイズしか使うことができません。

43/50 lines display：

EGA あるいは VGA が装備されている PC では，43/50 lines を選択すると，EGA システムでは43行×80桁，VGA システムでは50行×80桁の画面になります。

Options for editor メニュー

このメニューでは，統合開発環境のエディタの各種の機能のデフォルトを設定します。

Insert mode：

On/Off のトグルスイッチになっています。このオプションを On にすると，キーボードから打ち込んだ文字はカーソル位置に挿入され，カーソルよりも右側のテキストは右にずれていきます。また，このオプションを Off にすると，キーボードから入力された文字はカーソル位置に上書きされていきます。

Autoindent mode：

On/Offのトグルスイッチになっています。このオプションがOnの場合，**リターン**キーを押すと，カーソルは直前の行の一番始めの文字と同じ桁に移動します。Offのときは，カーソルは行の一番左の位置（1桁目）に戻ります。

Use tabs：

On/Offのトグルスイッチになっています。このオプションをOnにした場合，***TAB*** キーを押すと，エディタはTab sizeオプションで設定されているタブサイズを用いて，タブ文字（***CTRL-I***）をテキスト中に書き込みます。このオプションをOffにした場合，***TAB*** キーを押すと，直前の行の各語の先頭の位置までスペースを挿入します。

Optimal fill：

On/Offのトグルスイッチになっています。このオプションは，上のUse tabsオプションがOnのときだけ有効になります。この2つのオプションがOnの場合，行の先頭の空白部分はタブとスペースで"最適に"埋められます。これによって，各行の文字数（バイト数）は最少になります。

Backspace unindents：

On/Offのトグルスイッチになっています。このオプションをOnにすると，カーソルをアウトデントします。つまり，行の先頭の非空白文字，あるいは空行にカーソルがある場合に ***BS*** キーを押すと，カーソルは1つ左のインデントレベルへ戻ります。

Tab size：

このオプションを選択すると，プロンプトボックスが現われて，***TAB*** を押したときに挿入されるスペースの数を設定することができます。

Editor buffer size：

通常は小さいプログラムしか編集しない場合，エディタ用バッファを小さくしてデバッガに多くのメモリを与えることができます。このオプションを使うことによって，エディタバッファの大きさを20000～65534バイトの範囲で設定することができます。

Memory type of buffer(PC-9801のみ)：

エディタのバッファをどこに置くかを，Main memory，EMS，GVRAM bank #1の中から選択します。マシン（およびDOS）で64K以上のEMSメモリが利用可能な場合には，エディタバッファにEMSを使用して，プログラムのコンパイル・リンク・実

行用のメモリを64K ふやすことができます。また，(最初期型の PC-9801や PC-9801U2 などを除く）グラフィック VRAM を2バンク持っているマシンの場合には，エディタバッファを GVRAM のバンク#1に割り当てることもできます。ただしこの場合には，Mode for display を Text only にセットしておかなければなりません（したがって拡張23行および圧縮46行モードは使用できなくなります)。

Make use of EMS memory(IBM PC のみ)：

コンピュータに64K 以上の EMS メモリが含まれている場合，エディタが自動的に EMS をテキストバッファとして利用することができます。そうすると，64K バイトだけ RAM が解放されることになり，プログラムのコンパイル，リンク，実行時にそのメモリ領域を使うことができます。デフォルトは On となっています。このオプションを Off にすると，エディタは EMS メモリを使用しません。

Directories メニュー

Directories を使うと，TC.EXE の各デフォルトディレクトリのパスを設定することができます。ここで設定するディレクトリには，Turbo C がコンフィギュレーションファイル，ヘルプファイル，インクルードファイル，ライブラリをそれぞれ探しにいくディレクトリや，プログラムの出力されるディレクトリがあります。

Directories を選択するとサブメニューが表示されます。サブメニューの項目は次のとおりです。

- Include directories
- Library directories
- Output directory
- Turbo C directory
- Pick file name

各項目のパス名は，TC.EXE の対応するメニュー項目の場合と同じようにして入力します。項目の入力のしかたがよくわからない場合は第5章を参照してください。

Include directories

このオプションでは，Turbo C の標準インクルード（ヘッダ）ファイルが格納されているディレクトリを指定します。これを選択するとプロンプトボックスが現われるので，そ

こにディレクトリ名を入力します。

Include directories には，複数のディレクトリを入力することができます。各ディレクトリパス名はセミコロン（;）で区切ってください。最大127文字まで入力できます。また，絶対パス名と相対パス名のどちらを使ってもかまいません。

Library directories

このオプションでは，Turbo C のスタートアップオブジェクトファイル（C0*x*.OBJ）と実行時ライブラリファイル（.LIB）が格納されているディレクトリを指定します。これを選択するとプロンプトボックスが現われるので，そこにディレクトリ名を入力します。

Library directories には，複数のディレクトリを入力することができます。各ディレクトリパス名はセミコロン（；）で区切ってください。最大127文字まで入力できます。また，絶対パス名と相対パス名のどちらを使ってもかまいません。

例： `a:¥turboc¥lib; a:¥turboc¥mylibs; c:newturbo¥mathlibs; c:..¥vidlibs`

Output directory

このオプションでは，コンパイラが出力する.OBJ，.EXE，.MAP ファイルを書き込むデフォルトディレクトリを指定します。

Outpit directory には，ディレクトリパス名を1つ指定します。最大64文字まで入力可能です。

Turbo C directory

このオプションでは，ヘルプファイルと TCCONFIG.TC（デフォルトのコンフィギュレーションファイル）がカレントディレクトリになかった場合に，TC がそれらを探すディレクトリを指定します。

Turbo C directory には，ディレクトリパス名を1つ指定します。最大64文字まで入力可能です。

Pick file name

この項目を選択すると入力ウィンドウが現われるので，Turbo C がロードする（あるいは作成する）ピックアップのファイル名を入力してください。ピックファイルについては

第5章の第 III 部を参照してください。

Arguments

この項目では，プログラムを実行するときに渡されるコマンドライン引数のデフォルトを指定します。コマンドライン引数は，DOS のコマンドラインからタイプするのとまったく同じように入力します(ただしリダイレクションはサポートされません)。ここで指定するのは引数だけで，プログラム名はいりません。

Debug メニュー

Debug メニューの各項目を使うと，Turbo C 統合デバッガに対するデフォルト値を設定することができます。

Source debugging

デバッグを可能にするかどうかを選択します。このトグルを **On** にしてプログラムをコンパイルすると，統合デバッガあるいはスタンドアロンのデバッガを使用してデバッグが可能になります。Standalone にセットされている場合は，スタンドアロンのデバッガでしかデバッグできません。**None** にセットされている場合は，.EXE ファイルにはデバッグ情報は埋め込まれません。

Display swapping

このオプションを使うと，**Display** Swapping のデフォルトのレベルを **Smart**, **Always**, **None** のいずれかにセットすることができます。

これを **Smart** にセットしてデバッグモードでプログラムを走らせると，実行されるコードを見て，そのコードが画面に影響を与える（すなわち画面に出力する）かどうかを調べます。もしコードが画面に出力を行なう場合は，出力が行なわれる間だけ画面はエディタ画面から実行画面に切り換えられ，出力が終わると画面は元に戻ります。影響を与えない場合は，画面の切り換えは行なわれません。**Always** にセットされている場合は，1つの文を実行するたびに画面が切り換えられます。**None** にセットされている場合は，デバッガは画面の切り換えをまったく行ないません。

Editor commands オプション

Turbo C エディタは，次に示すような多くの編集コマンドを持っています。

- ■カーソル移動
- ■テキストの挿入と削除
- ■ブロック処理およびファイル処理
- ■文字列の検索（および置換）

こうした編集コマンドキー（およびキーの組み合わせ）に指定したいキーを割り当てることがます。編集コマンドについては付録 A で説明しています。

メインメニューから Editor commands を選ぶと Install Editor 画面が現われ，3列の項目が表示されます。

- ■最初（左）の列には，エディタで使用できる機能が表示されています。
- ■2番目の列は，第1キーストロークです。これによって，ある特定のエディタコマンドを呼び出すのにどんなキーや特殊キーの組み合わせを使うかを示しています。
- ■3番目の列は，第2キーストロークです。これによって，同じエディタコマンドを呼び出すもう1とつのキーストロークを用意することができます。第2キーストロークは省略してもかまいません。

注意：第2キーストロークは常に第1キーストロークより優先されます。

Install Editor 画面の一番下の行には，第1キーストローク，第2キーストロークの列の入力対象となる項目を選択するために使用するキーの要約が表示されています。

キー	機能	説明
↑↓←→	選択	キー割り当てを変更したい編集コマンドを選択します。
ROLLUP と ***ROLLDOWN***	ページ	1画面分上下にスクロールします。

リターン	変更	キーストローク変更モードに入ります。
R	初期設定	編集コマンドをすべて出荷時の設定に戻します。
ESC	終了	メインメニューに戻ります。
F4	キーモード	キーストロークの組み合わせの3つの方式を切り換えます。

リターンキーを押して変更モードに入ると，指定したコマンドに対して現在定義されているキーストロークがポップアップウィンドウに表示され，Install Editor 画面の一番下の行は，キーストロークの変更に使用するキーの要約に変わります。

キー	**機能**	**説明**
BS	後退	カーソルの左のキーストロークを削除します。
リターン	実行	指定された編集コマンドに対して新しく定義したキーストロークを割り付けます。
ESC	変更取消	現在選択されているコマンドに対する変更をキャンセルし，そのコマンドを元のキーストロークに戻して Install Editor 画面に帰ります（別の編集コマンドを選択できる状態になります）。
F2	回復	現在選択されているコマンドに対する変更をキャンセルし，元のキーストロークに戻します。ただし，選択中のコマンドはそのままで，再度定義を始めることができます。
F3	クリア	選択中のキーストローク定義をクリアします。クリアした後もキーストローク再定義状態のままです。

F4　キーモード　キーストローク組み合わせの3つの方式（WordStar-like, Ignore case, Verbatim）を切り換えます。

注意：エディタコマンドのキーストロークの中に ***F2***, ***F3***, ***F4*** キーを入力するときは，先にバッククォート（‘）を押してから ***F2***, ***F3*** などを押してください（PC-9801の最初期型やE/Fなど，バッククォートがないキーボードの場合には，かわりに ***SHIFT-*** ^を使ってください)。

キーストロークの組み合わせ方には，3つの方式(WordStar-like, Ignore case, Verbatim)があります。これは画面の一番下の行に表示されており，現在選択されている方式はハイライト表示されます。どの場合でも，キーの組み合わせの最初のキーは，特殊キーあるいは制御文字(***CTRL-文字***)である必要があります。キーの組み合わせ方式は，その後の文字をどう扱うかを示しています。

■ **WordStar-like**：このモードでは，英字や[,]，¥，^，_をタイプすると自動的に制御文字を入力したことになります。次のようになるわけです。

a, ***A***, ***CTRL-A*** をタイプすると，〈CTRL-A〉が入力される
y, ***Y***, ***CTRL-Y*** をタイプすると，〈CTRL-Y〉が入力される
[をタイプすると，〈CTRL-[〉が入力される

たとえば WordStar-like モードで，あるコマンドを〈CTRL-A〉〈CTRL-B〉に設定したとすると，TC エディタでは次のどれをタイプしてもそのコマンドが実行されます。

〈CTRL-A〉〈CTRL-B〉
〈CTRL-A〉B
〈CTRL-A〉b

■ **Ignore case**：このモードでは，入力した文字はすべて大文字に変換されます。ただし，文字を入力しても ***CTRL*** キーと文字の組み合わせには変換されません。したがって制御文字のキーストロークを入力する場合には，***CTRL*** キーを押しながら文字キーを押す必要があります。たとえばこのモードでは，〈CTRL-A〉B と〈CTRL-A〉b は同じですが，〈CTRL-A〉〈CTRL-B〉は前の2つとは異なります。

■ **Verbatim**：このモードでは，入力した文字はすべて正確に区別されます。たとえば，〈CTRL-A〉〈CTRL-B〉，〈CTRL-A〉B，〈CTRL-A〉b はすべて異なります。

使用できるキーストローク

Turbo C のエディタコマンドは，TCINST で好みに応じて自由にカスタマイズできるわけですが，キーストロークの定義には少しだけ制限があります。すべてのキーストロークモードに共通の制限もありますし，特定のモードにだけ存在する制限もあります。

1. 1つのエディタコマンドに対して与えることのできるキーストロークは最大6個までです。また，ある種のキー組み合わせはキーストローク2つに相当します。これには ***GRPH-文字***キー（***Alt-文字***），カーソル移動キー（↑，***ROLLUP***，***DEL***，***PgUp***），すべてのファンクションキーとの組み合わせ（***F4***，***SHIFT-F7***，***GRPH-F8*** など）があります。
2. 先頭のキーストロークには通常の文字（英字，カナ，数字）や記号は使えません。つまり，制御文字（***CTRL-文字***）か特殊キーでなければなりません。
3. コマンドのキーストロークとして ***ESC*** キーを入力するには，***CTRL-***[をタイプします。
4. コマンドのキーストロークとして ***BS*** キーを入力するには，***CTRL-H*** をタイプします。
5. コマンドのキーストロークとして**リターン**キーを入力するには，***CTRL-M*** をタイプします。
6. Turbo C で定義済みのヘルプファンクションキー（***F1***，***GRPH-F1***）は，Turbo C エディタコマンドキーに割り当てることはできません。これ以外のファンクションキーはすべて割り当てることができます。エディタコマンドキーの一部に Turbo C ホットキーを入力すると，TCINST はエディタ内でホットキーが無視されるとの警告を出し，そのキーを無視するかどうか確認してきます。Turbo C の定義済みホットキーの一覧表は第5章にあります。

Mode for display メニュー（PC-9801）

Turbo C 統合環境の画面表示に，テキスト画面のみを使うか，グラフィック画面も使用するかを指定します。

Text only（テキスト画面のみ）を選択すると，46行モードや16色表示はできなくなりま

すが，グラフィック画面を制御するルーチンの分のメモリを節約することができます。

Graphics and text（グラフィックとテキスト両方）を選択すると，グラフィック画面による拡張23行と圧縮46行モードを使用できるようになります。

Mode for display メニュー（IBM PC）

注意：このメニューは PC-9801用では存在しません。

通常は，Turbo C はユーザのシステムビデオモードを正しく検知します。ただし次のような場合には，Mode for display メニューを使って変更を行なってください。

- カレントビデオモード以外のモードを選択したい。
- "雪降り（snowy）"しないカラーグラフィックスアダプタを持っている。
- Turbo C がハードウエアを正しく検知していないと思われる。
- ラップトップシステムや，システムが単色の CGA のような動作をする合成画面をもっている。

メインメニューから Mode for display を選ぶと，サブメニューが現われます。統合環境で使用される画面モードをこのメニューから選択します。Default，Color，Black and white，LCD or composite，Monochrome の5つの選択肢があり，それぞれの意味は以下のとおりです。

Default

デフォルトでは，Turbo C がロードされたときにアクティブなビデオモードで動作します。

Color：

TC.EXE 起動時のアクティブモードに関わらず，カラーアダプタが検知されれば80桁カラーモードを使用します。終了するときには前のアクティブなモードに戻ります。

Black and white：

アクティブなモードに関わらず80桁白黒モードを使用し，終了すると前のアクティブモードに戻ります。このオプションは，ラップトップシステムや合成モニタの場合に必要になります。

LCD or composite：

ラップトップシステムや，システムが単色のCGAのような動作をする合成画面を持っている場合にはこれを選択してください。

Monochrome：

TC.EXE起動時のアクティブモードに関わらず，モノクロアダプタが検知されれば80桁モノクロモードを使用します。

最初の4つのどれかを選んだ場合，プログラムは画面のビデオテストを行ないます。何をするかを示す最下行のステータス行を参照してください。

どれかキーを押すと，ウィンドウには次の問い合わせが現われます。

```
Conducting video test.  Is your screen "snowy" now?
```

次のいずれかを選択することができます。

- **Yes**, ... 画面は"雪降り(snowy)"です。
- **No**, ... 雪降りのチェックは行ないません。
- **Maybe**, ... 必らずハードウエアをチェックします。

TCを複数のマシンで動作させたい場合は**Maybe**を選んでください。

メインメニューに戻るには***Esc***を押してください。

Set colors メニュー（PC-9801）

Turbo Cの表示色を変更する場合は，インストールのメインメニューからSを押します。**S**を押すと，まず次のようなオプションが表示されます。

- Customize colors
- Defalt color set
- Two tone color set
- Palette customize
- Initialize palette

Default color set を選ぶと，すべての部分の表示色を出荷時のデフォルトに設定することができます。

デフォルト以外に設定済みのカラーセットがもう1つ用意されています (**T**wo tone)。これは，特にモノクロディスプレイのために用意されているオプションで，階調表示のときにもはっきり区別できる色の組で構成されています(2色しか使用しないという意味ではありません)。

各部分ごとに表示色を設定する場合には，Customize colors を選択します。すると，メインメニュー，Edit ウィンドウ，Error ボックスなど，12個の画面項目からなるサブメニューが現われます。**A**～**L** の1つを押すか，矢印キーを使って項目の1つを選んでください。

画面項目で選ぶと，ポップアップメニューとビューポートが現われます。ビューポートは選択した項目の表示例で，ポップアップメニューにはその項目の構成要素が表示されます。

たとえば，Error ボックスの色を変更するために **H** を選択すると，ビューポート内では Error ボックスが手前に表示され，Error ボックスの4つの構成要素(**T**itle，**B**order，**N**ormal text，**H**ighlighted text）からなるポップアップメニューが現われます。

ポップアップメニューから要素の1つを選択すると，カラーパレットが現われます。矢印キーを使ってパレットから好みの色を選んでください。パレットの選択ボックスの移動につれて，ビューポートの中のその要素の色が変わっていきます。色が決まったら，リターンを押してその色を設定します。

この操作を，表示色を変更したいすべての画面項目に対して行ないます。終了したら，メインメニューに戻るまで ***ESC*** を押してください。

Palette customize を使うと，PC-9801のハードウェアパレット中の各色 (0～15) を直接設定することができます。これを選ぶと，現在のパレットの色を示す小さなウィンドウが現われます。変更したい色にカーソルを合せて**リターン**を押すと，RGB (Red, Green, Blue) の各色強度を指定するもうひとつのウィンドウが表示され，左右の矢印キーによって，0～15の範囲で色強度を設定することができます。色強度を変化させていくと，それにともなって画面上のそのパレット色の部分の色も次第に変わっていきます。

Palette customize で行なった設定内容(パレットごとの RGB の各色強度)は，TC.EXE に書き込まれます。そして，マシンをリセットした後でも，TC を起動すればパレットはその値にセットされます(ただし，TCINST の **O**ptions/**E**nvironment/**P**alette set up が Off にセットされている場合には，パレットはそのときのマシンの状態のままになります)。

Initialize palette は，ハードウェアパレットをマシンのデフォルト状態に設定します。**P**alette customize で行なった変更をすべてキャンセルしたいときなどに使用してくださ

い。

なお，ハードウェアパレットを操作するこの2つのオプションは，**Options/Environment/Screen size** で拡張23行（**Extended**）または圧縮46行（**High density**）にセットされている場合にしか選択できません。また，**Mode for display** が **Text only** にセットされている場合には，画面モードはテキスト画面による標準25行（**Standard**）のみとなり，したがってこの2つのオプションは選択できません。

Set colors メニュー(IBM PC)

Turbo C の表示色を変更する場合は，インストールのメインメニューから **S** を押します。**S** を押すと，まず次のようなオプションが表示されます。

- Customize colors
- Default color set
- Turquoise color set
- Version 1.x color set

Default color set を選ぶと，すべての部分の表示色を出荷時のデフォルトに設定することができます。

デフォルト以外に設定済みのカラーセットがもう1つ（Turquoise）用意されており，これが好みに合うものであれば，簡単に設定を済ませることができます。

また，**Version 1.x color set** を選ぶと，前のバージョンと同じ表示色が設定されます。

各部分ごとに表示色を設定する場合には，**Customize colors** を選択します。すると，メインメニュー，Edit ウィンドウ，Error ボックスなど，12個の画面項目からなるサブメニューが現われます。**A**～**L** の1つを押すか，矢印キーを使って項目の1つを選んでください。

画面項目で選ぶと，ポップアップメニューとビューポートが現われます。ビューポートは選択した項目の表示例で，ポップアップメニューにはその項目の構成要素が表示されます。

たとえば，Error ボックスの色を変更するために H を選択すると，ビューポート内では Error ボックスが手前に表示され，Error ボックスの4つの構成要素（**Title**，**Border**，**Normal text**，**Highlighted text**）からなるポップアップメニューが現われます。

ポップアップメニューから要素の1つを選択すると，カラーパレットが現われます。矢印キーを使ってパレットから好みの色を選んでください。パレットの選択ボックスの移動に

つれて，ビューポートの中のその要素の色が変わっていきます。色が決まったら，リターンを押してその色を設定します。

この操作を，表示色を変更したいすべての画面項目に対して行ないます。終了したら，メインメニューに戻るまで ***Esc*** を押してください。

注意：Turbo C は内部に3つのテーブルをもっています。それぞれ，カラー，白黒，モノクローム用のものです。TCINST では，現在のビデオモードに基づいて，一度にこれら3つの色の組のうち1つを変更することができます。したがって，たとえば白黒の色テーブルを変更したい場合は，ビデオモードを DOS プロンプトで BW80に設定してから，TCINST をロードしなければなりません。

Resize windows

このオプションを使うと，Turbo C の Edit ウィンドウと Message/Watch ウィンドウのサイズを変更することができます。メインメニューから ***R*** を押して **Resize windows** を選択してください。

上下の矢印キーを使って，Edit ウィンドウと Message ウィンドウの境界線を移動させることができます。ただし，どちらのウィンドウも1行未満にすることはできません。ウィンドウを好みのサイズに設定したら，**リターン**キーを押してください。ウィンドウサイズは，両ウィンドウの行数の比としてセーブされます。したがって，**O**ptions/**E**nvironment/Screen size で画面の行数を切り換えた場合，ウィンドウの境界線はここで設定したのとほぼ同じ場所に位置します。

ESC キーを押せば，変更を取りやめてメインメニューに戻ることができます。

プログラムの終了

必要な変更がすべて終わったら，メインメニューから Quit/save を選択してください。次のようなメッセージが画面の一番下に表示されます。

```
Save changes to TC.EXE? (Y/N)
```

■ ***Y***（Yes）を押すと，ここで行なった変更が永久的に Turbo C に組み込まれます（変更したければ TCINST を何度でも実行することができます）。

■ ***N***（No）を押すと，TCINST の中で行なった設定は無視され，Turbo C のデフォルトや起動時の表示は変更されずに MS-DOS プロンプトに戻ります。

Turbo C を出荷時のデフォルトに戻したいときは，単にマスターディスクから作業用のディスクに TC.EXE をコピーしてください。また，メインメニューで **E** オプションを選び，次に ***R***（Restore factory default）を押して ***ESC*** を押せば，エディタコマンドを元に戻すことができます。

付録 E

MicroCalc

MicroCalc は，Turbo C で記述されたスプレッドシート（表計算）プログラムです。そのソースコードとオブジェクトファイルは，サンプルプログラムとして Turbo C のマスターディスクの中に含まれています。このスプレッドシートプログラムは，テキストや数値，数式を入力する電子の紙のようなもので，入力されたものを自動的に計算します。

MicroCalc について

MicroCalc はデモンストレーションプログラムのため，以下のような制限事項があります（制限を取り除いていくのを楽しみにする人もいるでしょう）。

■数式をあるセルから他のセルにコピーすることはできません。
■テキストあるいは値をあるセルから他のセルにコピーすることはできません。
■合計を計算するセルは，同じ列または同じ行に並んでいるものでなければなりません。

こうした制限はありますが，MicroCalc には興味深い特徴がいくつかあります。

■表示を最高速にするために，ビデオ RAM に直接書き込みます。
■ひととおりの数学関数を備えています。
■テキストや数式を編集するためのラインエディタが組み込まれています。
■複数のセルにまたがって入力が行なえます。

このような特徴に加えて，MicroCalc はスプレッドシートとしての便利な機能を持っており，以下のようなことが行なえます。

■ディスクから表を読み込む。
■表をディスクにセーブする。
■何かを入力するたびに自動的に再計算される（再計算しないようにもできる）。
■表をプリンタで印刷する。
■現在の表の内容をクリアする。
■列または行を削除する。
■列の幅を設定する。
■すでにある行や列の間に空白行または空白列を挿入する。

MicroCalcのコンパイルと実行

MicroCalcのコンパイルは簡単です。まずマスターディスクから，MCALC.ARCファイルを作業用ディスクのTURBOCディレクトリ（TC.EXEやTCC.EXEが入っているディレクトリ）にコピーします。このファイルは圧縮形式になっているので，コンパイルする前に，マスターディスク中のUNPACK.COMを使って，次のようにしてソースファイルを復元しておきます。

```
unpack mcalc
```

MicroCalcは，Turbo Cのどちらのコンパイラでもコンパイルでき，また実行することができます。どちらのコンパイラを使う場合でも，ラージデータモデル（コンパクト，ラージ，ヒュージ）でコンパイルすれば，表計算で使用できるメモリが大きくなります。

TC.EXEの場合

Options/Directoriesメニューでインクルードとライブラリのディレクトリを設定したら，次のことを行ないます。

1. TC.EXEを起動する。
2. **Project**メニューで，プロジェクト名（**Project** name）にMCALC.PRJを指定する。
3. メインメニューから**Run**/**Run**コマンドを選択する。

注意：コンパイルの際にファイルハンドルを使いきってしまうことがあるかもしれないので，ルートディレクトリの CONFIG.SYS ファイルに FILES=20（あるいはそれ以上）を書き込んで，使用できるハンドルの数を大きくしておいてください。これは TCC の場合でも同じです。

TCC.EXE の場合

MS-DOS のプロンプトから次のコマンドを入力します。

```
TCC mcalc mcparser mcdisplay mcinput mcommand mcutil
```

注意：コマンドラインオプション -I と -L によって，インクルードとライブラリディレクトリを必要に応じて指定しなければなりません。

MicroCalc の使い方

MicroCalc のコンパイルがすんだら，これを実行するには 2 つの方法があります。

TC の Run/Run コマンドでコンパイルすれば，画面に MicroCalc が現われ，MicroCalc を終了すれば Turbo C に戻ります。

MS-DOS のコマンドラインから MCALC.EXE を動かしたい場合は，ただ MCALC とタイプします。もうすでにデータファイルがある場合には，次のようにタイプすると，そのファイルが自動的にロードされます。

```
MCALC <your_file>
```

次に示したのは，MicroCalc をロードしたときの画面例です。

```
           A        B        C        D        E        F
1        22.00
2         1.00
3         2.00
4         3.00
5        28.00

...

20
A5 Formula
A1+A2+A3+A4
```

MicroCalc の画面は，たくさんのセルにわかれています。**セル**は，列と行のペアによって特定される領域（マス目）です。列の名前と行番号の組み合わせが**セル座標**になります。デフォルトでは，各列の幅は10桁になっています。この幅は，最大77桁までの間で変更することができます。

列は，A～Z，AA～CV の範囲（100列），行は1～100の範囲があり，合計で10000セルあることになります。ヘッダファイル MCALC.H の中の定数 MAXROWS と MAXCOLS を修正することによって，この制限を変更することができます。

セルには，数値，数式，テキスト（文字列）を入力することができ，これらはセルのタイプになっています。セルのタイプと座標（列-行）は，画面の最下行の左端に表示されます。

```
A5 Formula    現在のセルはA5で，数式が入っている
A1 Text       A1のセルにはテキストが入ってる
A2 Value      A2のセルには数値が入っており，他のセルを参照してはいない
```

前の画面例の A5 Formula という行は，アクティブなセルは A5で，そこには数式が入っていることを表しています。最後の行 A1＋A2＋A3＋A4は，アクティブなセルには，A1から A4までの合計が入っていることを示しています。この２行は，A1，A2，A3，A4のセルの数値を合計して，その結果を A5に置くことを意味します。

この数式は，A1：A4という省略形で書くことができ，“A1から A4までのすべてのセルの値を加算する”ことを表わします。

セルを参照する有効な数式の例を示しておきましょう。

```
A1+(B2-C7)    B2からC7の値を引き，結果をA1に加える
A1:A23        A1，A2，A3，...，A23のすべてのセルの合計
```

数式は，必要に応じて複雑なものにすることができます。たとえば次のようにもなります。

```
SIN(A1)*COS(A2)/((1.2*A8)+LOG(ABS(A8)+8.9E-3))+(C1:C5)
```

セルにデータを入力するには，目的のセルのカーソルを動かしてから，必要な文字をタイプします。MicroCalc は，入力されたデータが数値か，数式か，テキストかを自動的に判定します。数式はテキストとしてセルに格納されますが，それが数式とみなせる場合にはその計算の結果が表示されるようになっています。

MicroCalc の標準の関数と演算子

+ - * /	加算，減算，乗算，除算
^	べき乗（2^3 = 8 など）
:	セルのグループの合計（A1：A4 = A1+A2+A3+A4など）
ABS	絶対値
ACOS	アークコサイン（逆余弦）
ASIN	アークサイン（逆正弦）
ATAN	アークタンジェント（逆正接）
COS	コサイン（余弦）
COSH	ハイパボリックコサイン（双曲線余弦）
EXP	指数関数
LOG	自然対数
LOG10	常用対数
POW10	10の累乗
ROUND	もっとも近い整数に丸める
SIN	サイン（正弦）
SINH	ハイパボリックサイン（双曲線正弦）
SQR	2乗
SQRT	平方根
TAN	タンジェント（正接）
TANH	ハイパボリックタンジェント（双曲線正接）
TRUNC	数値の整数部を返す

MicroCalcの標準コマンド

/	メインメニューを呼び出す
/SL	表のデータファイルをロードする
/SS	現在の表をセーブする
/SP	現在の表を印刷する
/SC	現在の表をクリアする
/F	セルのグループを書式化する
/D	現在のセルを削除する
/G	指定のセルにカーソルを移動する
/CI	列を挿入する
/CD	現在の列を削除する
/CW	現在の列の幅を変更する
/RI	行を挿入する
/RD	現在の行を削除する
/E	現在のセルを編集する
/UR	表の中の数式を再計算する
/UF	数式そのものと，その結果のどちらを表示するかを切り換える
/A	自動再計算のオン/オフを切り換える
/Q	MicroCalcを終了する
DEL	現在のセルを削除する
HOME	A1のセルに移動する
HELP (***End***)	一番右下のセルに移動する
ROLLUP (***PgUp***) または ***CTRL-*** ↑, ***SHIFT-*** ↑	1画面上へ
ROLLDOWN (***PgDn***) または ***CTRL-*** ↓, ***SHIFT-*** ↓	1画面下へ
←, →, ↑, ↓	セルカーソルを上下左右に動かす
F2	現在のセル中のデータを編集する

また，入力行の編集中は以下のキーが使えます。

ESC	データに加えた変更をキャンセルする
←, →	カーソルを左右に動かす
↑, ↓, リターン	編集を終えてデータをセルに入れる
HOME	カーソルを入力行の先頭に移動する
HELP（***End***）	カーソルを入力行の最後に移動する
DEL	カーソル位置の文字を削除する
INS	挿入/上書きモードを切り換える
BS	カーソルの左の文字を削除する

MicroCalc のパーサ

ここに示す情報は，MicroCalc のパーサ（parser：構文解析ルーチン）を変更したいという人のために提供されています(たとえば，2つの引数をとる関数を追加したいような場合です)。このパーサの state（状態）と goto の情報は，UNIX の yacc ユーティリティを使って作成されました。yacc への入力は以下のとおりです。

```
%token CONST CELL FUNC
%%
e : e '+' t
  | e '-' t
  | t
  ;
t : t '*' f
  | t '/' f
  | f
  ;
f : x '^' f
  | x
  ;
x : '-' u
  | u
  ;
u : CELL ':' CELL
  | o
  ;
o : CELL
  | '(' e ')'
  | CONST
  | FUNC '(' e ')'
  ;
%%
```

索　引

【A】

【B】

【C】

【D】

【E】

【F】

【G】

【H】

【N】

【O】

【U】

【V】

【W】

【X】

【Z】

【あ】

【か】

【さ】

【な】

【は】

【ま】

【や】

【ら】

【わ】

Turbo C 2.0 ユーザーズガイド

1988 年 12 月 20 日　初版発行

著者　Borland International

編訳・発行　(株)マイクロソフトウェア アソシエイツ

〒107 東京都港区南青山 7-8-1 小田急南青山ビル

電話 (03) 486-1411 (代表)

(03) 486-1403 (サポートセンター)

FAX (03) 486-8905

Printed in Japan

落丁, 乱丁はお取り替えいたします。

TURBO C®

USER'S GUIDE

BORLAND